

В. Юрлов
ASSEMBLER

В учебнике рассматриваются вопросы программирования на языке ассемблера для компьютеров на базе микропроцессоров фирмы Intel. Основу книги составляет материал, являющийся частью курса, читаемого автором в высшем учебном заведении и посвященного вопросам системного программирования. По сравнению с предыдущей книгой автора — «Assembler: учебный курс», книга существенно переработана. В нее добавлены разделы, посвященные программированию под Windows, и описанию команд процессоров Pentium III.

К книге прилагается дискета, которая содержит все листинги программ, рассматриваемых в учебнике, и соответствующий пояснительный материал.

Для студентов вузов, программистов и всех желающих изучить язык Assembler.

Краткое содержание

| | |
|--|-----|
| Предисловие | 9 |
| Урок 1. Общие сведения об ЭВМ | 14 |
| Урок 2. Архитектура персонального компьютера | 26 |
| Урок 3. Простая программа на ассемблере | 57 |
| Урок 4. Жизненный цикл программы на ассемблере | 66 |
| Урок 5. Структура программы на ассемблере | 86 |
| Урок 6. Система команд микропроцессора | 111 |
| Урок 7. Команды обмена данными | 133 |
| Урок 8. Арифметические команды | 153 |
| Урок 9. Логические команды | 184 |
| Урок 10. Команды передачи управления | 202 |
| Урок 11. Цепочечные команды | 229 |
| Урок 12. Сложные структуры данных | 250 |
| Урок 13. Макросредства языка ассемблера | 278 |
| Урок 14. Модульное программирование | 310 |
| Урок 15. Прерывания | 353 |
| Урок 16. Защищенный режим работы микропроцессора | 380 |
| Урок 17. Обработка прерываний в защищенном режиме. | 404 |
| Урок 18. Создание Windows-приложений на ассемблере | 419 |
| Урок 19. Архитектура и программирование сопроцессора | 500 |
| Урок 20. MMX-технология микропроцессоров Intel | 564 |
| Вместо заключения | 623 |

Содержание

| | |
|--|----|
| Предисловие | 9 |
| Урок 1. Общие сведения об ЭВМ | 14 |
| Урок 2. Архитектура персонального компьютера | 26 |
| Архитектура ЭВМ | 27 |
| Набор регистров | 38 |

| | |
|--|-----|
| Организация памяти | 46 |
| Типы данных | 51 |
| Формат команд | 54 |
| Обработка прерываний | 55 |
| Урок 3. Простая программа на ассемблере | 57 |
| Урок 4. Жизненный цикл программы на ассемблере | 66 |
| Трансляция программы | 71 |
| Компоновка программы | 77 |
| Отладка программы | 79 |
| Утилита MAKE | 83 |
| Урок 5. Структура программы на ассемблере | 86 |
| Синтаксис ассемблера | 88 |
| Директивы сегментации | 98 |
| Простые типы данных ассемблера | 105 |
| Урок 6. Система команд микропроцессора | 111 |
| Системы счисления | 112 |
| Двоичная система счисления | 113 |
| Шестнадцатеричная система счисления | 114 |
| Десятичная система счисления | 116 |
| Перевод чисел из одной системы счисления в другую | 116 |
| Перевод в десятичную систему счисления | 116 |
| Перевод в двоичную систему счисления | 116 |
| Перевод в шестнадцатеричную систему счисления | 117 |
| Перевод дробных чисел | 118 |
| Числа со знаком | 121 |
| Структура машинной команды | 123 |
| Способы задания операндов команды | 126 |
| Функциональная классификация машинных команд | 131 |
| Урок 7. Команды обмена данными | 133 |
| Пересылка данных | 135 |
| Ввод-вывод в порт | 137 |
| Работа с адресами и указателями | 143 |
| Преобразование данных | 145 |
| Работа со стеком | 147 |
| Урок 8. Арифметические команды | 153 |
| Общий обзор | 154 |
| Целые двоичные числа | 155 |
| Десятичные числа | 157 |
| Арифметические операции над целыми двоичными числами | 159 |
| Сложение двоичных чисел без знака | 159 |
| Сложение двоичных чисел со знаком | 160 |
| Вычитание двоичных чисел без знака | 162 |
| Вычитание двоичных чисел со знаком | 164 |

| | |
|---|-----|
| Вычитание и сложение операндов большой размерности | 165 |
| Умножение двоичных чисел без знака | 166 |
| Умножение двоичных чисел со знаком | 168 |
| Деление двоичных чисел без знака | 168 |
| Деление двоичных чисел со знаком | 170 |
| Вспомогательные команды для целочисленных операций | 170 |
| Команды преобразования типов | 170 |
| Другие полезные команды | 172 |
| Арифметические операции над двоично-десятичными числами | 173 |
| Неупакованные BCD-числа | 174 |
| Упакованные BCD-числа | 180 |
| Урок 9. Логические команды | 184 |
| Логические данные | 186 |
| Логические команды | 187 |
| Команды сдвига | 191 |
| Линейный сдвиг | 192 |
| Циклический сдвиг | 194 |
| Дополнительные команды сдвига | 196 |
| Примеры работы с битовыми строками | 198 |
| Рассогласование битовых строк | 198 |
| Вставка битовых строк | 199 |
| Извлечение битовых строк | 200 |
| Пересылка битов | 200 |
| Урок 10. Команды передачи управления | 202 |
| Безусловные переходы | 208 |
| Команда безусловного перехода jmp | 208 |
| Процедуры | 212 |
| Условные переходы | 217 |
| Команда сравнения cmp | 218 |
| Команды условного перехода и флаги | 220 |
| Команды условного перехода и регистр ecx/cx | 222 |
| Организация циклов | 222 |
| Урок 11. Цепочечные команды | 229 |
| Пересылка цепочек | 233 |
| Команда movs | 234 |
| Команды пересылки байтов, слов и двойных слов | 235 |
| Сравнение цепочек | 236 |
| Команда cmps | 236 |
| Команды сравнения байтов, слов и двойных слов | 239 |
| Сканирование цепочек | 239 |
| Команда scas | 240 |
| Сканирование строки байтов, слов, двойных слов | 242 |
| Загрузка элемента цепочки в аккумулятор | 242 |

| | |
|---|------------|
| Команда lods | 242 |
| Загрузка в регистр al/ax/eah байтов, слов, двойных слов | 244 |
| Перенос элемента из аккумулятора в цепочку | 244 |
| Команда stos | 244 |
| Сохранение в цепочке байта, слова, двойного слова из регистра al/ax/eah | 246 |
| Ввод элемента цепочки из порта ввода-вывода | 247 |
| Вывод элемента цепочки в порт ввода-вывода | 247 |
| Урок 12. Сложные структуры данных | 250 |
| Массивы | 252 |
| Описание и инициализация массива в программе | 252 |
| Доступ к элементам массива | 254 |
| Типовые операции с массивами | 261 |
| Структуры | 264 |
| Описание шаблона структуры | 265 |
| Определение данных с типом структуры | 265 |
| Методы работы со структурой | 266 |
| Объединения | 268 |
| Записи | 270 |
| Описание записи | 271 |
| Определение экземпляра записи | 271 |
| Работа с записями | 273 |
| Дополнительные возможности обработки | 275 |
| Урок 13. Макросредства языка ассемблера | 278 |
| Псевдооператоры equ и = | 280 |
| Макрокоманды | 283 |
| Макродирективы | 291 |
| Директивы WHILE и REPT | 292 |
| Директива IRP | 293 |
| Директива IRPC | 294 |
| Директивы условной компиляции | 294 |
| Директивы компиляции по условию | 295 |
| Директивы генерации ошибок | 302 |
| Константные выражения в условных директивах | 305 |
| Дополнительное управление трансляцией | 307 |
| Урок 14. Модульное программирование | 310 |
| Технологии программирования | 311 |
| Структурное программирование | 312 |
| Концепция модульного программирования | 313 |
| Процедуры в языке ассемблера | 315 |
| Организация интерфейса с процедурой | 317 |
| Связь ассемблера с языками высокого уровня | 330 |
| Связь Pascal—ассемблер | 333 |

| | |
|---|------------|
| Команды enter и leave | 343 |
| Связь С—ассемблер | 346 |
| Урок 15. Прерывания | 353 |
| Контроллер прерываний | 357 |
| Программирование контроллера прерываний 18259A | 362 |
| ICW1 — определить особенности последовательности приказов | 363 |
| ICW2 — определение базового адреса | 363 |
| ICW3 — связь контроллеров | 364 |
| ICW4 — дополнительные особенности обработки прерываний | 366 |
| OCW1 — управление регистром масок IMR | 366 |
| OCW2 — управление приоритетом | 366 |
| OCW3 — общее управление контроллером | 367 |
| Каскадирование микросхем i8259A | 368 |
| Реальный режим работы микропроцессора | 369 |
| Обработка прерываний в реальном режиме | 371 |
| Урок 16. Защищенный режим работы микропроцессора | 380 |
| Системные регистры микропроцессора | 382 |
| Регистры управления | 383 |
| Регистры системных адресов | 384 |
| Регистры отладки | 385 |
| Структуры данных защищенного режима | 386 |
| Пример программы защищенного режима | 391 |
| Подготовка таблиц глобальных дескрипторов GDT | 392 |
| Запрет обработки аппаратных прерываний | 397 |
| Переключение микропроцессора в защищенный режим | 398 |
| Работа в защищенном режиме | 399 |
| Переключение микропроцессора в реальный режим | 401 |
| Разрешение прерываний | 402 |
| Стандартное для MS-DOS завершение работы программы | 402 |
| Урок 17. Обработка прерываний в защищенном режиме | 404 |
| Шлюз ловушки | 411 |
| Шлюз прерывания | 412 |
| Шлюз задачи | 413 |
| Инициализация таблицы IDT | 415 |
| Обработчики прерываний | 416 |
| Программирование контроллера прерываний 8259A | 416 |
| Загрузка регистра IDTR | 417 |
| Урок 18. Создание Windows-приложений на ассемблере | 419 |
| Каркасное Windows-приложение на C/C++ | 423 |
| Каркасное Windows-приложение на ассемблере | 435 |
| Стартовый код (строки 54-73) | 445 |
| Главная функция (строки 74-162) | 447 |
| Обработка сообщений в оконной функции | 456 |

| | |
|--|-----|
| Средства TASM для разработки Windows-приложений | 459 |
| Углубленное программирование на ассемблере для Win32 | 462 |
| Ресурсы Windows-приложений на языке ассемблера | 463 |
| Меню в Windows-приложениях | 464 |
| Перерисовка изображения | 472 |
| Окна диалога в Windows-приложениях | 480 |
| Урок 19. Архитектура и программирование сопроцессора | 500 |
| Архитектура сопроцессора | 502 |
| Регистр состояния swr | 507 |
| Регистр управления cwr | 508 |
| Регистр тегов twr | 509 |
| Форматы данных | 509 |
| Двоичные целые числа | 510 |
| Упакованные целые десятичные (BCD) числа | 511 |
| Вещественные числа | 512 |
| Специальные численные значения | 516 |
| Система команд сопроцессора | 520 |
| Команды передачи данных | 522 |
| Команды загрузки констант | 525 |
| Команды сравнения данных | 525 |
| Арифметические команды | 528 |
| Команды трансцендентных функций | 537 |
| Команды управления сопроцессором | 549 |
| Исключения сопроцессора и их обработка | 554 |
| Исключение недействительная операция | 555 |
| Деление на ноль | 556 |
| Денормализация операнда | 556 |
| Переполнение и антипереполнение | 556 |
| Неточный результат | 556 |
| Немаскируемая обработка исключений | 557 |
| Использование отладчика | 559 |
| Общие рекомендации по программированию сопроцессора | 561 |
| Урок 20. MMX-технология микропроцессоров Intel | 564 |
| MMX-расширение архитектуры микропроцессора Pentium | 566 |
| Модель целочисленного MMX-расширения | 566 |
| Особенности команд MMX-расширение | 568 |
| Система команд | 574 |
| Отладка программ | 578 |
| Пример применения MMX-технологии | 593 |
| Дополнительные целочисленные MMX-команды (Pentium III) | 606 |
| ХММ-расширение архитектуры микропроцессора Pentium | 608 |
| Модель XMM-расширения | 609 |
| Система команд | 611 |

Предисловие

Эта книга посвящена одному из самых старых из существующих сегодня языков программирования — ассемблеру. Интересно проследить, начиная со времени появления первых компьютеров и заканчивая сегодняшним днем, за трансформациями представлений об этом языке у программистов. Когда-то это был основной язык, без знания которого нельзя было заставить компьютер сделать что-либо полезное. Постепенно ситуация менялась. Появлялись более удобные средства общения с компьютером. Но в отличие от других языков ассемблер не умирал, более того, он не мог сделать этого в принципе. Почему? Чтобы ответить на этот вопрос, нужно понять, что такое язык ассемблера. Если коротко, то ассемблер — это символическое представление машинного языка. Все процессы в машине на самом низком, аппаратном уровне приводятся в действие только командами (инструкциями) машинного языка. Отсюда понятно, что, несмотря на общее название, язык ассемблера для каждого типа компьютера свой. Это касается и внешнего вида программ, написанных на ассемблере, и идей, отражением которых этот язык является. По-настоящему решить проблемы, связанные с аппаратурой, невозможно без знания ассемблера. Программист или любой другой пользователь может использовать любые высокоуровневые средства, вплоть до программ построения виртуальных миров и, возможно, даже не подозревать, что на самом деле компьютер выполняет не команды языка, на котором написана его программа, а их трансформированное представление в форме скучной и унылой последовательности команд совсем другого языка — машинного. А теперь представим, что у такого пользователя возникла нестандартная проблема или просто что-то не получается. К примеру, его программа должна работать с некоторым необычным устройством или выполнять другие действия, связанные с непосредственным обращением к аппаратуре. И вот здесь-то и начинается «совсем другая история». Каким бы умным не был программист, каким бы хорошим не был язык, на котором он написал свою чудную программу, без знания ассемблера ему не обойтись. И не случайно практически все компиляторы языков высокого уровня содержат средства связи своих модулей с модулями на ассемблере либо поддерживают выход на ассемблерный уровень программирования. Конечно, время компьютерных универсалов уже прошло. Как говорится «нельзя обять необъятное». Но есть нечто общее в базовой подготовке всех программистов, своего рода фундамент, — это знание принципов работы компьютера, его архитектуры и языка ассемблера, отражающего устройство компьютера. Без рассмотрения данных вопросов невозможно любое сколько-нибудь серьезное компьютерное образование.

В книге рассматриваются вопросы программирования на языке ассемблера для компьютеров на базе микропроцессоров фирмы Intel. Основу книги составляет материал, являющийся частью курса, читаемого автором в высшем учебном заведении и посвященного вопросам системного программирования. Это наложило отпечаток не только на методику изложения материала, но и позволило расставить необходимые акценты на тех вопросах, которые обычно вызывают трудности у студентов. Но научить – это только одна цель книги. Вторая цель книги – стать хорошим справочным пособием по языку ассемблера. Ведь мало знать набор и назначение команд. Смысл многих команд далеко неочевиден, а некоторые из них имеют свойства, которыми можно воспользоваться в ситуациях, когда команда применяется не по своему прямому назначению. Для таких команд приводятся алгоритмы, в контексте которых эти команды используются. Знание таких особенностей и их воплощение в своей программе может вызвать истинное восхищение у ваших оппонентов. И ради этого стоит потратить время на оптимальную реализацию вашей идеи в форме компьютерной программы.

Эта книга адресуется следующим категориям читателей:

- молодым людям, школьникам, углубленно изучающим программирование для компьютеров на базе микропроцессоров Intel;
- студентам вузов, готовящимся стать профессиональными программистами и изучающим архитектуру микропроцессоров Intel с его языком ассемблера в рамках соответствующих дисциплин;
- специалистам, профессионально занимающимся программированием и желающим освоить ассемблер для решения стоящих перед ними задач;
- всем, кто интересуется программированием микропроцессоров Intel на низком уровне или просто желает познакомиться с тем, как устроен и работает компьютер.

Так как ассемблер является символическим представлением машинного языка, то он неразрывно связан с архитектурой самого микропроцессора. По мере внесения изменений в его архитектуру совершенствуется и язык ассемблера. По этой причине книга стремится решить комплексную задачу – не просто рассмотреть ассемблер как еще один из сотен языков программирования, а показать неразрывную связь его конструкций с архитектурой микропроцессора. Материал книги содержит описание основных особенностей архитектур и системы команд микропроцессоров Pentium Pro/MMX/II/III.

Изложение материала ведется в форме уроков. Логически книга делится на три части.

Первая часть книги описывает основы языка ассемблера. Она состоит из одиннадцати уроков.

На первых двух уроках читатель узнает, что представляет собой современный компьютер, что включают в себя понятия архитектуры микропроцессора и компьютера в целом. При рассмотрении этого материала становится очевидной роль языка ассемблера как выразителя архитектуры компьютера.

На третьем и четвертом уроках читатель познакомится с типичной программой на языке ассемблера и поймет, что представляет собой «ассемблерный» уровень

программирования. Читатель также познакомится со средствами создания используемых модулей. Кроме того, на четвертом уроке читатель узнает об отладочных средствах, которые помогут ему выйти из затруднительных положений, когда программа, написанная на ассемблере, отказывается работать.

На пятом и шестом уроках читатель узнает, как правильно оформить программу на ассемблере, и познакомится с ее синтаксическими конструкциями. В конце шестого занятия читатель познакомится с классификацией целочисленных машинных команд, в соответствии с которой будет вестись их обсуждение на последующих уроках (уроки 7–11).

Вторая часть книги, начиная с урока 12, посвящена углубленному изучению вопросов программирования с использованием языка ассемблера. Так, на уроке 12 читатель подробно познакомится со средствами ассемблера для работы со структурами данных, которые характерны для языков высокого уровня (таких как Pascal и C). Это несколько приближает уровень программирования на ассемблере к указанным языкам.

На уроке 13 читатель очень подробно познакомится с весьма полезным инструментом языка ассемблера — макросредствами. Макросы, при надлежащем овладении ими, могут сделать процесс программирования на ассемблере не только легким, но и приятным.

Урок 14 посвящен очень важному вопросу — организации модульного программирования с использованием ассемблера. Подробно описываются все тонкости связи отдельных программ, написанных на ассемблере. Затем показывается, что эти принципы действительны и при связывании программ на ассемблере с программами на других языках. Понятно, что описать все возможные случаи просто невозможно, тем более, что многое здесь зависит от особенностей (и даже версии) конкретного компилятора языка высокого уровня. Но тем не менее в основе такой связи лежат несколько основных основополагающих принципов, понимание которых позволит читателю быстрее сориентироваться в конкретной ситуации.

Уроки 15–17 рассматривают режимы работы микропроцессора, организацию его взаимодействия с остальными устройствами компьютера. Данная информация, возможно, не будет востребована немедленно, но она позволит читателю осмысленно подходить к вопросу программирования на компьютере, даже и без использования языка ассемблера.

Третью часть книги составляют уроки 18–20.

Урок 18 рассматривает процесс разработки оконных приложений для Windows. Освоив материал данного урока, вы научитесь создавать полноценные Windows-приложения, которые будут не хуже, чем те, что написаны традиционным способом, — на языке высокого уровня. Консольные Windows-приложения также не забыты, но их разработка не представляет особых трудностей, поэтому они рассматриваются на примере конкретной задачи в уроке 20.

Урок 19 рассматривает вопросы программирования устройства с плавающей точкой (сопроцессора). Это дополнительное устройство процессора позволяет значительно повысить удобство и качество программирования вычислительных задач. Сопроцессор достаточно часто используется для расчетов в задачах машинной

графики. Поэтому в данном уроке рассматривается пример Windows-приложения, производящего графический вывод, рассчитанный сопроцессором.

Урок 20 посвящен рассмотрению особенностей архитектуры и программирования MMX-расширения микропроцессора Pentium. Впервые это расширение появилось в архитектуре микропроцессора Pentium MMX. Это было целочисленное MMX-расширение. В определенной степени оно не оправдало надежд разработчиков, но, тем не менее, включало ряд полезных свойств. Фирма Intel продолжила это направление развития своих микропроцессоров, и последняя модель ее микропроцессора Pentium III содержит модуль MMX-расширения с плавающей точкой. Система команд микропроцессора Pentium III включает более 360 команд. Понятно, что даже упрощенный их перечень займет достаточно много места. Если вас интересует подробная информация о командах процессора Intel Pentium III и других процессоров семейства Intel x86, обратитесь к моей книге «Assembler: специальный справочник», выпускаемой издательством «Питер».

В этом справочнике вы найдете детальную информацию о командах ассемблера с примерами их использования. Кроме того, справочник содержит дополнительный материал, который будет полезен для практической работы. Справочник органично дополняет учебник, а вместе эти две книги составят хороший комплект для тех, кто всерьез решил заняться разработкой ассемблерных программ.

К книге прилагается дискета, которая содержит все основные программы книги и соответствующий пояснительный материал. Хотелось бы надеяться, что содержащиеся на дискетах программы значительно повысят удобство и гибкость работы с материалом книги.

Что еще нужно для работы с книгой? Во многом это зависит от целей, которые читатель ставит перед собой. Если это простое знакомство, то компьютер может и не понадобиться, для этого достаточно прочитать нужный материал. Но научиться программировать на ассемблере таким образом, конечно, нельзя. Для этого нужен компьютер на базе одного из микропроцессоров Intel, начиная с i8086 и заканчивая Pentium. Оптимальный выбор — Pentium II/III. Из программного обеспечения необходим редактор текстов, вплоть до самого простого (notepad.exe), и пакет транслятора ассемблера фирм Microsoft или Borland. Для того чтобы разговор с читателем шел на одном языке, ему желательно отдать предпочтение транслятору TurboAssembler (TASM) фирмы Borland версии 5.0, потому, что именно он использовался для разработки программ из книги. Ну и конечно, необходимо большое желание, терпение и тяга к познанию, так как первые шаги всегда бывают трудными. Сложность еще и в том, что материал книги неразрывно связан со множеством других вопросов, касающихся не только компьютерного «железа». Понятно, что учесть и рассмотреть их все в рамках одной книги просто невозможно, да и вряд ли нужно. С этой точки зрения читателю следует рассматривать книгу, скорее, как описание инструмента и приемов работы с ним для решения некоторых конкретных задач.

Быстрому появлению книги на свет, в частности первого ее издания, способствовали некоторые студенты, среди которых надо отдельно отметить Алексея Пашкова, Антона Попача и Артема Букаева. Особую благодарность выражают моим жене Елене и детям — Саше и Юле. Они стойко переносили все тяготы жизни,

связанные с постоянной занятостью компьютера и невозможностью из-за этого поработать и поиграть в любимые игры. Хотелось бы выразить благодарность также всем остальным окружающим меня людям, кто активно или пассивно помогал появлению этой книги на свет.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter-press.ru (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на Web-сайте издательства <http://www.piter-press.ru>.



1

УРОК

Общие сведения об ЭВМ

-
- Путешествие в историю, далекую и не очень
 - Внешний вид типичного современного компьютера
 - Структурная схема компьютера
 - Что такое ассемблер?
-

Современному человеку сегодня трудно представить свою жизнь без **электронно-вычислительных машин** (ЭВМ). В настоящее время любой желающий, в соответствии со своими запросами, может собрать у себя на рабочем столе полноценный вычислительный центр. Так было, конечно, не всегда. Путь человечества к этому достижению был труден и тернист. Много веков назад люди хотели иметь приспособления, которые помогали бы им решать разнообразные задачи. Многие из этих задач решались последовательным выполнением некоторых рутинных действий, или, как принято говорить сейчас, выполнением алгоритма. С попытки изобрести устройство, способное реализовать простейшие из этих алгоритмов (сложение и вычитание чисел), все и началось...

Точной отсчета можно считать начало XVII века (1623 год), когда ученый В. Шиккард создал машину, умеющую складывать и вычитать числа. Но первым *арифмометром*, способным выполнять четыре основных арифметических действия, стал арифмометр знаменитого французского ученого и философа Блеза Паскаля. Основным элементом в нем было зубчатое колесо, изобретение которого уже само по себе стало ключевым событием в истории вычислительной техники. Правнуки этого колеса еще совсем недавно, каких-нибудь полтора десятка лет назад, использовались в арифмометрах (соответствующая модель была создана в 1842 году) на столах советских бухгалтеров. Хотелось бы отметить, что эволюция в области вычислительной техники носит неравномерный, скачкообразный характер: периоды накопления сил сменяются прорывами в разработках, после чего наступает период стабилизации, во время которого достигнутые результаты используются практически и одновременно накапливаются знания и силы для очередного рывка вперед. После каждого витка процесс эволюции выходит на новую, более высокую ступень.

В 1671 году немецкий философ и математик Густав Лейбниц также создает арифмометр на основе зубчатого колеса особенной конструкции — *зубчатого колеса Лейбница*. Арифмометр Лейбница, как и арифмометры его предшественников, выполнял четыре основных арифметических действия. На этом данный период закончился, и человечество в течение почти полутора веков копило силы и знания для следующего витка эволюции вычислительной техники. XVIII и XIX века были временем, когда бурно развивались различные науки, в том числе математика и астрономия. В них часто возникали задачи, требующие длительных и трудоемких вычислений.

Еще одним известным человеком в истории вычислительной техники стал английский математик Чарльз Бэббидж. В 1823 году Бэббидж начал работать над машиной для вычисления полиномов, но, что более интересно, эта машина

должна была, кроме непосредственного производства вычислений, выдавать результаты — печатать их на негативной пластине для фотопечати. Планировалось, что машина будет приводиться в действие паровым двигателем. Из-за технических трудностей Бэббиджу до конца не удалось реализовать свой проект. Здесь впервые возникла идея использовать некоторое *внешнее (периферийное) устройство* для выдачи результатов вычислений. Отметим, что другой ученый, С. Шойц, в 1853 году все же реализовал машину, задуманную Бэббиджем (она получилась даже меньше, чем планировалась). Наверное, Ч. Бэббиджу больше нравился творческий процесс поиска новых идей, чем воплощение их в нечто материальное. В 1834 году он изложил принципы работы очередной машины, которая была названа им «аналитической». Технические трудности вновь не позволили ему до конца реализовать свои идеи. Бэббидж смог довести машину лишь до стадии эксперимента. Но именно идея является двигателем научно-технического прогресса. Очередная машина Чарльза Бэббиджа была воплощением следующих идей:

- *Управление производственным процессом.* Машина управляла работой ткацкого станка, изменяя узор создаваемой ткани в зависимости от сочетания отверстий на специальной бумажной ленте. Эта лента стала предшественницей таких знакомых нам всем носителей информации, как перфокарты и перфоленты.
- *Программируемость.* Работой машины также управляла специальная бумажная лента с отверстиями. Порядок следования отверстий на ней определял команды и обрабатываемые этими командами данные. Машина имела арифметическое устройство и память. В состав команд машины входила даже команда условного перехода, изменяющая ход вычислений в зависимости от некоторых промежуточных результатов.

В разработке этой машины принимала участие графиня Ада Августа Лавлейс¹, которую считают первой в мире *женщиной-программистом*.

Не слишком ли много для одного проекта и одного человека?!

Идеи Чарльза Бэббиджа развивались и использовались другими учеными. Так, в 1890 году, на рубеже XX века, американец Г. Холлерит разработал машину, работающую с таблицами данных (первый Excel?). Машина управлялась программой на перфокартах. Она использовалась при проведении переписи населения в США в 1890 году. В 1896 году Г. Холлерит основал фирму, явившуюся предшественницей корпорации IBM. Со смертью Бэббиджа в эволюции вычислительной техники наступил очередной перерыв вплоть до 30-х годов XX века. В дальнейшем все развитие человечества стало немыслимым без компьютеров.

В 1938 году центр разработок ненадолго смещается из Америки в Германию, где К. Цузе создает машину, которая оперирует, в отличие от своих предшественниц, не десятичными числами, а двоичными. Эта машина также была все еще механической, но ее несомненным достоинством было то, что в ней была реализована идея обработки данных в *двоичном коде*. Продолжая свои работы, Цузе в 1941 году создал электромеханическую машину, арифметическое устройство которой было выполнено на базе *rеле*. Машина умела выполнять операции с плавающей точкой.

¹ В честь графини Ады Августы Лавлейс, родственницы Байрона, был назван язык программирования Ada.

За океаном, в Америке, в этот период также шли работы по созданию подобных электромеханических машин. В 1944 году Г. Айкен спроектировал машину, которую называли MARK-1. Она, как и машина К. Цузе, работала на реле. Но из-за того, что эта машина явно была создана под влиянием работ Бэббиджа, она оперировала с данными в десятичной форме.

Естественно, из-за большого удельного веса механических частей эти машины были обречены. Нужно было искать новую, более технологичную элементную базу. И тогда вспомнили об изобретении Л. Фореста, который в 1906 году создал трехэлектродную вакуумную лампу, названную *триодом*. В силу своих функциональных свойств она стала наиболее естественной заменой реле. В 1946 году в США, в университете города Пенсильвания, была создана *первая универсальная ЭВМ* — ENIAC. ЭВМ ENIAC содержала 18 тыс. ламп, весила 30 тонн, занимала площадь 200 м² и потребляла огромную мощность. В ней все еще использовались десятичные операции, и программирование осуществлялось путем коммутации разъемов и установки переключателей. Естественно, что такое «программирование» влекло за собой появление множества проблем, вызванных, прежде всего, неверной установкой переключателей. С проектом ENIAC связано имя еще одной ключевой фигуры в истории вычислительной техники — математика Джона фон Неймана. Именно он впервые предложил записывать программу и ее данные в память машины так, чтобы их можно было при необходимости модифицировать в процессе работы. Этот ключевой принцип, получивший название *принципа хранимой программы*, был использован в дальнейшем при создании принципиально новой ЭВМ EDVAC (1951 год). В этой машине уже применяется двоичная арифметика и используется оперативная память, построенная на ультразвуковых ртутных линиях задержки. Память могла хранить 1024 слова. Каждое слово состояло из 44 двоичных разрядов.

После создания EDVAC человечество осознало, какие высоты науки и техники могут быть достигнуты tandemом человек—компьютер. Данная отрасль стала развиваться очень быстро и динамично, хотя здесь тоже наблюдалась некоторая периодичность, связанная с необходимостью накопления определенного багажа знаний для очередного прорыва. До середины 80-х годов процесс эволюции вычислительной техники принято делить на поколения. Для полноты изложения дадим этим поколениям краткие качественные характеристики:

1-е поколение (1945–1954 гг.) — время становления машин с фон-неймановской архитектурой. В этот период формируется типовой набор структурных элементов, входящих в состав ЭВМ. К этому времени у разработчиков уже сложилось примерно одинаковое представление о том, из каких элементов должна состоять типичная ЭВМ. Это — *центральный процессор* (ЦП), *оперативная память* (или *оперативное запоминающее устройство* — ОЗУ) и *устройства ввода-вывода* (УВВ). ЦП, в свою очередь, должен состоять из *арифметико-логического устройства* (АЛУ) и *управляющего устройства* (УУ). Машины этого поколения работали на ламповой элементной базе, из-за чего поглощали огромное количество энергии и были очень ненадежны. С их помощью, в основном, решались научные задачи. Программы для этих машин уже можно было составлять не на машинном языке, а на языке ассемблера.

2-е поколение (1955–1964 гг.). Смену поколений определило появление новой элементной базы: вместо громоздкой лампы в ЭВМ стали применяться миниатюрные

транзисторы, линии задержки как элементы оперативной памяти сменила память на магнитных сердечниках. Это в конечном итоге привело к уменьшению габаритов, повышению надежности и производительности ЭВМ. В архитектуре ЭВМ появились индексные регистры и аппаратные средства для выполнения операций с плавающей точкой. Были разработаны команды для вызова подпрограмм. Появились языки высокого уровня — Algol, FORTRAN, COBOL, — создавшие предпосылки для появления переносимого программного обеспечения, не зависящего от типа ЭВМ. С появлением языков высокого уровня возникли компиляторы для них, библиотеки стандартных подпрограмм и другие хорошо знакомые нам сейчас вещи. Важное новшество, которое хотелось бы отметить, — это появление так называемых *процессоров ввода-вывода*. Эти специализированные процессоры позволили освободить ЦП от управления вводом-выводом и осуществлять ввод-вывод с помощью специализированного устройства одновременно с процессом вычислений. На этом этапе резко расширился круг пользователей ЭВМ и возросла номенклатура решаемых задач. Для эффективного управления ресурсами машины стали использоваться *операционные системы* (ОС).

3-е поколение (1965–1970 гг.). Смена поколений вновь была обусловлена обновлением элементной базы: вместо транзисторов в различных узлах ЭВМ стали использоваться *интегральные микросхемы* различной степени интеграции. Микросхемы позволили разместить десятки элементов на пластине размером в несколько сантиметров. Это, в свою очередь, не только повысило производительность ЭВМ, но и снизило их габариты и стоимость. Появились сравнительно недорогие и малогабаритные машины — *мини-ЭВМ*. Они активно использовались для управления различными технологическими производственными процессами в системах сбора и обработки информации. Увеличение мощности ЭВМ сделало возможным одновременное выполнение нескольких программ на одной ЭВМ. Для этого нужно было научиться координировать между собой одновременно выполняемые действия, для чего были расширены функции операционной системы. Одновременно с активными разработками в области аппаратных и архитектурных решений растет удельный вес разработок в области технологий программирования. В это время активно разрабатываются теоретические основы методов программирования, компиляции, баз данных, операционных систем и т. д. Создаются пакеты прикладных программ для самых различных областей жизнедеятельности человека. Теперь уже становится непозволительной роскошью переписывать все программы с появлением каждого нового типа ЭВМ. Наблюдается тенденция к созданию семейств ЭВМ, то есть машины становятся совместимы снизу вверх на программно-аппаратном уровне. Примерами таких семейств была серия IBM System 360 и наш отечественный аналог — ЕС ЭВМ.

4-е поколение (1970–1984 гг.). Очередная смена элементной базы привела к смене поколений. В 70-е годы активно ведутся работы по созданию больших и сверхбольших интегральных схем (БИС и СБИС), которые позволили разместить на одном кристалле десятки тысяч элементов. Это повлекло дальнейшее существенное снижение размеров и стоимости ЭВМ. Работа с программным обеспечением стала более дружественной, что повлекло за собой рост количества пользователей. В принципе, при такой степени интеграции элементов стало возможным попытаться создать функционально полную ЭВМ на одном кристалле. Соответствующие попытки были предприняты, хотя они и встречались, в основном, не-

доверчивой улыбкой. Наверное, этих улыбок стало бы меньше, если бы можно было предвидеть, что именно эта идея станет причиной «вымирания» больших ЭВМ через каких-нибудь полтора десятка лет. Тем не менее в начале 70-х годов фирмой Intel был выпущен микропроцессор (МП) i4004. И если до этого в мире вычислительной техники были только три направления (суперЭВМ, большие ЭВМ (мэйнфреймы) и мини-ЭВМ), то теперь к ним прибавилось еще одно — микропроцессорное. В общем случае под *процессором* понимают функциональный блок ЭВМ, предназначенный для логической и арифметической обработки информации на основе *принципа микропрограммного управления*. По аппаратной реализации процессоры можно разделить на микропроцессоры (полностью интегрирующие все функции процессора) и процессоры с малой и средней интеграцией. Конструктивно это выражается в том, что микропроцессоры реализуют все функции процессора на одном кристалле, а процессоры других типов реализуют их путем соединения большого количества микросхем.

Итак, первый МП i4004 был создан фирмой Intel на рубеже 70-х годов. Он представлял собой 4-разрядное параллельное вычислительное устройство, и его возможности были сильно ограничены. i4004 мог производить четыре основные арифметические операции и применялся поначалу только в карманных калькуляторах. Позднее сфера его применения была расширена за счет использования в различных системах управления (например, для управления светофорами). Фирма Intel, правильно предугадав перспективность микропроцессоров, продолжила интенсивные разработки, и один из ее проектов в конечном итоге привел к крупному успеху, предопределившему будущий путь развития вычислительной техники. Им стал проект по разработке 8-разрядного микропроцессора i8008 (1972 г.). Этот микропроцессор имел довольно развитую систему команд и умел делить числа. Именно он был использован при создании персонального компьютера Альтаир, для которого молодой Билл Гейтс написал один из своих первых интерпретаторов языка Basic. Наверное, именно с этого момента следует вести отсчет 5-го поколения.

5-е поколение можно назвать микропроцессорным. Заметьте, что 4-е поколение закончилось только в начале 80-х, то есть «родители» в лице больших машин и их быстро взрослеющее и набирающее силы «чадо» в течение почти 10 лет относительно мирно существовали вместе. Для них обоих это время пошло только на пользу. Проектировщики больших компьютеров накопили огромный теоретический и практический опыт, а программисты микропроцессоров сумели найти свою, пусть поначалу очень узкую, нишу на рынке. В 1976 году фирма Intel закончила разработку 16-разрядного микропроцессора i8086. Он имел достаточно большую разрядность регистров (16 бит) и системной шины адреса (20 бит), за счет чего мог адресовать до 1 Мбайт оперативной памяти. В 1982 году был создан i80286. Этот микропроцессор представлял собой улучшенный вариант i8086. Он поддерживал уже несколько режимов работы: *реальный*, когда формирование адреса производилось по правилам i8086, и *защищенный*, который аппаратно реализовывал многозадачность и управление виртуальной памятью. i80286 имел также большую разрядность шины адреса — 24 разряда против 20 у i8086, и поэтому он мог адресовать до 16 Мбайт оперативной памяти. Первые компьютеры на базе этого микропроцессора появились в 1984 году. По своим вычислительным возможностям этот компьютер стал сопоставим с IBM 370. Поэтому можно считать, что на этом 4-е поколение развития ЭВМ завершилось.

В 1985 году фирма Intel представила первый 32-разрядный микропроцессор i80386, аппаратно совместимый снизу вверх со всеми предыдущими микропроцессорами этой фирмы. Он был гораздо мощнее своих предшественников, имел 32-разрядную архитектуру и мог прямо адресовать до 4 Гбайт оперативной памяти. Микропроцессор i386 стал поддерживать новый режим работы – режим виртуального i8086, который обеспечил не только большую эффективность работы программ, разработанных для i8086, но и позволил осуществлять параллельную работу нескольких таких программ. Еще одно важное нововведение – поддержка страничной организации оперативной памяти – позволило иметь виртуальное пространство памяти размером до 4 Тбайт (терабайт). Микропроцессор i386 был первым микропроцессором, в котором использовалась параллельная обработка. Так, одновременно осуществлялись: доступ к памяти и устройствам ввода-вывода, размещение команд в очереди для выполнения, их декодирование, преобразование линейного адреса в физический, а также страничное преобразование адреса (информация о 32-х наиболее часто используемых страницах помещалась в специальную кэш-память).

Вскоре после микропроцессора i386 появился i486. В его архитектуре получили дальнейшее развитие идеи параллельной обработки. Устройство декодирования и исполнения команд было организовано в виде пятиступенчатого конвейера, на котором в различной стадии исполнения могло находиться до 5 команд. На кристалл была помещена кэш-память первого уровня, которая содержала часто используемые код и данные. Кроме этого, появилась кэш-память второго уровня емкостью до 512 Кбайт. Появилась возможность строить многопроцессорные конфигурации. В систему команд процессора были добавлены новые команды. Все эти нововведения, наряду со значительным (до 133 МГц) повышением тактовой частоты микропроцессора, значительно позволили повысить скорость выполнения программ.

С 1993 года стали выпускаться микропроцессоры Intel Pentium. Их появление вначале омрачилось ошибкой в блоке операций с плавающей точкой. Эта ошибка была быстро устранена, но недоверие к этим микропроцессорам еще некоторое время оставалось. Pentium продолжил развитие идей параллельной обработки. В устройство декодирования и исполнения команд был добавлен второй конвейер. Теперь два конвейера (называемых *u* и *v*) вместе могли исполнять две инструкции за такт. Внутренний кэш был увеличен вдвое – до 8 Кбайт для кода и 8 Кбайт для данных. Процессор стал более интеллектуальным. В него была добавлена возможность предсказания ветвлений, в связи с чем значительно возросла эффективность исполнения нелинейных алгоритмов. Несмотря на то что архитектура системы оставалась все еще 32-разрядной, внутри микропроцессора стали использоваться 128- и 256-разрядные шины передачи данных. Внешняя шина данных была увеличена до 64 бит. Продолжили свое развитие технологии, связанные с многопроцессорной обработкой информации.

Появление микропроцессора Pentium Pro разделило рынок на два сектора – высокопроизводительных рабочих станций и дешевых домашних компьютеров. В микропроцессоре Pentium Pro были реализованы самые передовые технологии. В частности был добавлен еще один конвейер к имевшимся двум у микропроцессора Pentium. Тем самым за один такт работы микропроцессор стал выполнять до трех инструкций. Более того, микропроцессор Pentium Pro позволил осуществлять

лять динамическое исполнение команд (Dynamic Execution). Суть его в том, что три устройства декодирования команд, работая параллельно, делят команды на более мелкие части, называемые микрооперациями (micro-ops). Далее эти микрооперации могут исполняться параллельно пятью устройствами (двумя целочисленными, двумя с плавающей точкой и одним устройством интерфейса с памятью). На выходе эти инструкции опять собираются в первоначальном виде и порядке. Мощь Pentium Pro дополняется усовершенствованной организацией его кэш-памяти. Как и процессор Pentium, он имеет 8 Кбайт кэш-памяти первого уровня и 256 Кбайт кэш-памяти второго уровня. Однако за счет схемных решений (использование архитектуры двойной независимой шины) кэш-память второго уровня расположили на одном кристалле с микропроцессором, что значительно повысило производительность. В Pentium Pro реализовали 36-разрядную адресную шину, что позволило адресовать до 64 Гбайт оперативной памяти.

Процесс развития семейства обычных микропроцессоров Pentium тоже не стоял на месте. Если в микропроцессорах Pentium Pro параллелизм вычислений был реализован за счет архитектурных и схемотехнических решений, то при создании моделей микропроцессора Pentium пошли по другому пути. В них включили новые команды, для поддержки которых несколько изменили программную модель микропроцессора. Эти команды, получившие название MMX-команд (MultiMedia eXtention — мультимедийное расширение системы команд), позволили одновременно обрабатывать несколько единиц однотипных данных.

На этом эксперименты с делением рынка на сектора фирма Intel пока закончила и предпочла другой путь, суть которого заключалась в том, что модель конкретного микропроцессора «весила» столько, чтобы удовлетворить потребности пользователей в конкретном секторе рынка. Следующий выпущенный в свет микропроцессор, названный Pentium II, объединил в себе все технологические достижения обоих направлений развития архитектуры Pentium. Кроме этого он имел новые конструктивные особенности, в частности, его корпус выполнен в соответствии с новой технологией изготовления корпусов. Не забыт и рынок портативных компьютеров, в связи с чем микропроцессором поддерживаются несколько режимов энергосбережения.

Ну и наконец, последняя новинка фирмы Intel — микропроцессор Pentium III. Традиционно он поддерживает все достижения своих предшественников, главное (и, возможно, единственное?) его достоинство — наличие новых 70 команд. Эти команды дополняют группу MMX-команд, но для чисел с плавающей точкой. Для поддержки этих команд в архитектуру микропроцессора был включен специальный блок.

Таким образом, несомненные преимущества, заложенные в архитектуре микропроцессоров Pentium, помогли сохранить ему лидирующие позиции на российском рынке и стать самыми массовыми и популярными у нас в стране.

В этой книге мы подробно будем обсуждать архитектуру и особенности программирования именно микропроцессоров Intel. Мы сознательно опустили обсуждение микропроцессоров других фирм. Они, несомненно, имеют свои достоинства (и недостатки), которые в силу специфики книги здесь обсуждать неуместно.

Типичный современный компьютер (на базе микропроцессора Pentium) состоит из компонентов, показанных на рис. 1.1.

Системный блок



Рис. 1.1. Компьютер и периферийные устройства

Из рисунка видно, что компьютер состоит из нескольких физических устройств, каждое из которых подключено к одному блоку, называемому *системным*. Если рассуждать логически, то ясно, что он играет роль некоторого координирующего устройства. Давайте заглянем внутрь системного блока (не нужно пытаться проникнуть внутрь монитора — там нет ничего интересного, к тому же это опасно): открываем корпус и видим какие-то платы, блоки, соединительные провода. Чтобы понять их функциональное назначение, рассмотрим структурную схему типичного современного компьютера. Она не претендует на безусловную точность и имеет целью лишь показать назначение, взаимосвязь и типовой состав элементов современного персонального компьютера.

На рис. 1.2 показана схема компьютера на базе микропроцессоров семейства Intel P6, к которым относятся Pentium Pro/II/III, поэтому она немного, но не принципиально, отличается от схем компьютеров на базе более ранних моделей микропроцессоров. На схеме представлены: центральный процессор, оперативная память, внешние устройства. Все компоненты соединены между собой через системную шину. Системная шина имеет дополнительную шину — шину расширения. В компьютерах на базе i486 и Pentium в качестве такой шины используется шина PCI (Peripheral Component Interface), к которой подсоединяются внешние устройства, а также шины более ранних стандартов (например ISA — Industry Standard Architecture).

На схеме показана самая общая схема сердца компьютера — микропроцессора. Основу микропроцессора составляют блок микропрограммного управления, исполнительное устройство, регистры. Остальные компоненты микропроцессора выполняют вспомогательные функции. Более подробный вариант этой схемы мы рассмотрим на следующем уроке.

Наша книга посвящена именно вопросам работы процессора и его взаимоотношениям с другими компонентами компьютера.

Чтобы лучше понять принципы работы компьютера, давайте сравним его с человеком. У компьютера есть органы восприятия информации из внешнего мира — это клавиатура, мышь, накопители на магнитных дисках. На рис. 1.2 эти органы расположены под системными шинами. У компьютера есть органы, «переварива-

ющие» полученную информацию, — это центральный процессор и оперативная память. И наконец, у компьютера есть органы речи, выдающие результаты переработки. Это также некоторые из устройств ввода/вывода, расположенные в нижней части рис. 1.2. Современным компьютерам, конечно, далеко до человека. Их можно сравнить с существами, взаимодействующими с внешним миром на уровне большого, но ограниченного набора безусловных рефлексов. Этот набор рефлексов образует *систему машинных команд*. На каком бы высоком уровне вы ни общались с компьютером, в конечном итоге все сводится к скучной и однообразной последовательности машинных команд. Каждая машинная команда является своего рода раздражителем для возбуждения того или иного безусловного рефлекса. Реакция на этот раздражитель всегда однозначная и «защита» в блоке микропрограммного управления в виде микропрограммы. Эта микропрограмма и реализует действия по выполнению машинной команды, но уже на уровне сигналов, подаваемых на те или иные логические схемы компьютера, тем самым управляя различными подсистемами компьютера. В этом состоит так называемый принцип микропрограммного управления. Продолжая аналогию с человеком, отметим: для того, чтобы компьютер правильно «питался», придумано множество операционных систем, компиляторов сотен языков программирования и т. д. Но все они являются, по сути, лишь блюдом, на котором по определенным правилам доставляется пища (программы) желудку (компьютеру). Только (вот досада!) желудок компьютера любит диетическую, однообразную пищу — подавай ему информацию структурированную, в виде строго организованных последовательностей нулей и единиц, комбинации которых и составляют *машинный язык*. Таким образом, внешне являясь полиглотом, компьютер понимает только один язык — язык машинных команд. Конечно, для общения и работы с компьютером необязательно знать этот язык, но практически любой профессиональный программист рано или поздно сталкивается с необходимостью его изучения. К счастью, программисту не нужно пытаться постичь значение различных комбинаций двоичных чисел, так как еще в 50-е гг. программисты стали использовать для программирования символический аналог машинного языка, который называли *языком ассемблера*. Этот язык точно отражает все особенности машинного языка. Именно поэтому, в отличие от языков высокого уровня, язык ассемблера для каждого типа компьютеров свой.

Из всего вышесказанного можно сделать вывод, что, так как язык ассемблера для компьютера «родной», то и самая эффективная программа может быть написана только на нем (при условии, что ее пишет квалифицированный программист). Здесь есть одно маленькое «но»: это очень трудоемкий и требующий большого внимания и практического опыта процесс. Поэтому реально на ассемблере пишут в основном только программы, которые должны обеспечить эффективную работу с аппаратной частью. Иногда на ассемблере пишутся критичные по времени выполнения или расходованию памяти участки программы. Впоследствии они оформляются в виде подпрограмм и совмещаются с кодом на языке высокого уровня. На наших уроках в дальнейшем мы подробно разберемся с большинством перечисленных выше областей применения ассемблера.

Если вы держите в руках эту книгу, значит, для вас настало время сделать очередной шаг в профессиональном росте. Поэтому переворачивайте страницы и приступайте к уроку 2.

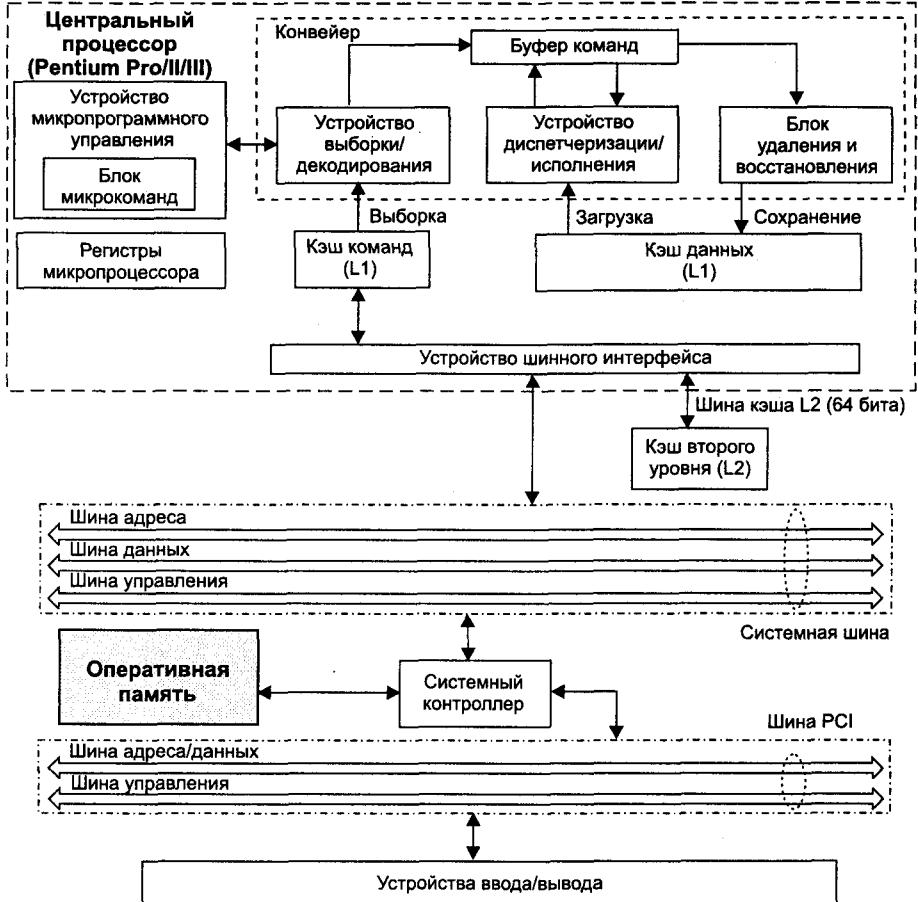


Рис. 1.2. Структурная схема персонального компьютера

Подведем некоторые итоги:

- Очень нелегок и длителен был путь развития вычислительной техники. Вначале были простые машины, выполняющие несложные арифметические действия. Люди постарше помнят широко распространенные полтора десятка лет назад механические арифмометры, изобретенные еще в XVII веке, и 30-тонные машины с очень ограниченными возможностями в конце 50-х годов. И вот в конце XX века мы имеем компактный 30-килограммовый набор устройств с колоссальными потенциальными возможностями.
- Несмотря на большие различия во внешнем виде, структурно все компьютеры устроены примерно одинаково. В их состав обязательно входят центральный процессор, внешняя и оперативная память, устройства ввода-вывода и отображения информации.

- Работать компьютер заставляет некий «серый кардинал» — машинный язык. Пользователь может даже и не подозревать о его существовании. Общаться с компьютером пользователю помогают операционные системы, офисные пакеты, системы программирования и т. д. Использование современных технологий программирования позволяет создавать программы, не написав ни строчки кода. Но в мозг компьютера команды все же поступают на машинном языке.
- Машинный язык полностью отражает все архитектурные тонкости конкретного типа компьютеров. Следствием этого является то, что он индивидуален для каждого семейства ЭВМ. Для того чтобы использовать эффективно все возможности компьютера, применяют символический аналог машинного языка — язык ассемблера.
- Работать на компьютере можно и без знания языка ассемблера. Но элементом подготовки программиста-профессионала обязательно является изучение ассемблера. Почему? Изучая ассемблер, вы обязательно попутно познакомитесь с архитектурой компьютера. А это, в свою очередь, позволит вам в дальнейшем создавать более эффективные программы на других языках и объединять их, при необходимости, с программами на ассемблере.



2

УРОК

Архитектура персонального компьютера

-
- Понятие об архитектуре ЭВМ
 - Архитектурные особенности компьютеров на базе i486 и Pentium
 - Описание набора регистров микропроцессора
 - Организация оперативной памяти компьютера
 - Форматы и типы данных, поддерживаемые микропроцессором
 - Формат машинных команд
 - Система прерываний компьютера
-

На уроке 1 мы описали компьютер на «житейском» уровне. При обсуждении понятия машинного языка отмечалось, что его характеристики полностью определяются особенностями того типа компьютера, для которого этот язык предназначен. Возникает вопрос — как оценить возможности конкретного типа (или модели) компьютера и его отличительные особенности от компьютеров других типов (моделей). Рассмотрения одной лишь только его структурной схемы явно недостаточно, так как она принципиально мало чем отличается для разных машин. У всех компьютеров есть оперативная память, процессор, внешние устройства. Различными являются способы, средства и используемые ресурсы, с помощью которых компьютер функционирует как единый механизм. Чтобы собрать воедино все понятия, характеризующие компьютер с точки зрения его функциональных программно-управляемых свойств, существует специальный термин — архитектура ЭВМ. Впервые это понятие стало упоминаться с появлением машин 3-го поколения для их сравнительной оценки. Мы отмечали на уроке 1, что в это время наблюдался всплеск разработок как программного, так и аппаратного обеспечения. Ниже мы дадим более формальное определение понятия архитектуры ЭВМ. Но прежде хотелось бы предупредить читателя, что урок будет нелегким. Приступить к изучению языка ассемблера любого компьютера имеет смысл только после выяснения того, какая часть компьютера оставлена видимой и доступной для программирования на этом языке. Это так называемая *программная модель* компьютера. Для программирования на языках высокого уровня совсем необязательно вникать слишком глубоко в эти вопросы. Мы не стали «размазывать» вопросы архитектуры по всей книге, а систематизированно изложили их на одном уроке. На последующих уроках эти вопросы будут уточняться и расширяться, но у вас уже будет перед глазами общая картина. Если же вы чувствуете, что перестаете понимать материал, то просто просмотрите его и переходите к уроку 3. В любом случае в процессе работы над книгой периодически возвращайтесь к этому уроку и со временем вы найдете ответы на многие вопросы.

Архитектура ЭВМ

Это понятие довольно трудно определить однозначно, потому что при желании в него можно включить все, что связано с ЭВМ вообще и какой-то конкретной моделью компьютера в частности. Попытаемся все же formalизовать этот широко распространенный термин.

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

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

Как видите, понятие архитектуры включает в себя практически всю необходимую для программиста информацию о компьютере. Поэтому прежде чем приступить к изучению вопросов, связанных с программированием на ассемблере для компьютеров на базе микропроцессоров фирмы Intel, познакомимся с их архитектурой.

Все современные ЭВМ обладают некоторыми общими и индивидуальными свойствами архитектуры. Индивидуальные свойства присущи только конкретной модели компьютера и отличают ее от больших и малых собратьев. Наличие общих архитектурных свойств обусловлено тем, что большинство типов существующих машин принадлежат 4 и 5 поколениям ЭВМ так называемой фон-неймановской архитектуры. К числу *общих архитектурных свойств и принципов* можно отнести:

- *Принцип хранения программы.* Согласно ему, код программы и ее данные находятся в одном адресном пространстве в оперативной памяти.
- *Принцип микропрограммирования.* Суть этого принципа заключается в том, что машинный язык все-таки еще не является той конечной субстанцией, которая физически приводит в действие процессы в машине. В состав процессора входит блок *микропрограммного управления* (см. рис. 1.2). Этот блок для каждой машинной команды имеет набор действий-сигналов, которые нужно генерировать для физического выполнения требуемой машинной команды. Здесь уместно вспомнить характеристику ЭВМ 1-го поколения. В них для генерации нужных сигналов необходимо было осуществить ручное программирование всех логических схем — поистине адская и неблагодарная работа!
- *Линейное пространство памяти* — совокупность ячеек памяти, которым последовательно присваиваются номера (адреса) 0, 1, 2,
- *Последовательное выполнение программ.* Процессор выбирает из памяти команды строго последовательно. Для изменения прямолинейного хода выполнения программы или осуществления ветвления необходимо использовать специальные команды. Они называются командами условного и безусловного перехода.
- С точки зрения процессора, нет принципиальной разницы между данными и командами. Данные и машинные команды находятся в одном пространстве памяти в виде последовательности нулей и единиц. Это свойство связано

с предыдущим. Процессор, исполняя содержимое некоторых последовательных ячеек памяти, всегда пытается трактовать его как коды машинной команды, а если это не так, то происходит аварийное завершение программы, содержащей некорректный фрагмент. Поэтому важно в программе всегда четко разделять пространство данных и команд.

- *Безразличие к целевому назначению данных.* Машине все равно, какую логическую нагрузку несут обрабатываемые ею данные.

Наша книга посвящена вопросам программирования микропроцессоров фирмы Intel – i486 и Pentium¹. У них, как и у процессоров других фирм, есть индивидуальные архитектурные принципы. К слову сказать, перечень архитектурных нововведений для этих микропроцессоров впечатляет. Их полное рассмотрение не является нашей целью, поэтому уделим внимание наиболее характерным и необходимым для дальнейшего изложения новациям.

Суперскалярная архитектура. Для того чтобы пояснить этот термин, разберемся вначале со значением другого термина – *конвейеризация вычислений*. Важным элементом архитектуры, появившимся в i486, стал *конвейер* – специальное устройство, реализующее такой метод обработки команд внутри микропроцессора, при котором исполнение команды разбивается на несколько этапов. i486 имеет пятиступенчатый конвейер. Соответствующие пять этапов включают:

- выборку команды из кэш-памяти или оперативной памяти;
- декодирование команды;
- генерацию адреса, при которой определяются адреса operandов в памяти;
- выполнение операции с помощью АЛУ;
- запись результата (куда будет записан результат, зависит от алгоритма работы конкретной машинной команды).

Таким образом, на стадии выполнения каждая машинная команда как бы разбивается на более элементарные операции. В чем преимущество такого подхода? Очередная команда после ее выборки попадает в блок декодирования. Таким образом, блок выборки свободен и может выбрать следующую команду. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Скорость вычисления в результате существенно возрастает. Микропроцессоры, имеющие один конвейер, называются *скалярными*, а два и более – *суперскалярными*. Микропроцессор Pentium имеет два конвейера, то есть использует суперскалярную архитектуру, и поэтому может выполнять две команды за машинный такт. Внутренняя структура конвейера такая же, как и у i486. Микропроцессоры семейства P6 (Pentium Pro/II/III) имеют другую структуру конвейера. Так как это наиболее перспективные микропроцессоры, то мы рассмотрим их структуру более подробно. Отметим, что описываемая в данной книге система команд полностью поддерживается именно этими микропроцессорами.

¹ Этот выбор обусловлен тем, что указанные процессоры являются наиболее популярными на сегодняшний день в России. Все программы в данной книге приведены для этих микропроцессоров. Так как i486 и Pentium являются результатом эволюции более ранних моделей микропроцессоров фирмы Intel, то они полностью совместимы с предыдущими моделями микропроцессоров. Исходя из этого, большую часть сведений, приведенных в книге, можно использовать для программирования младших моделей микропроцессоров Intel.

Раздельное кэширование кода и данных. Кэширование — это способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в так называемой «кэш-памяти первого уровня» (быстрой памяти), находящейся внутри микропроцессора. i486, к примеру, содержит один блок встроенной кэш-памяти размером 8 Кбайт, который используется для кэширования и кодов, и данных. Pentium содержит два блока кэш-памяти: один для кода и один для данных, каждый по 8 Кбайт. При этом становится возможным одновременный доступ к коду и данным, что увеличивает скорость работы компьютера.

Предсказание правильного адреса перехода. Под *переходом* понимается запланированное алгоритмом изменение последовательного характера выполнения программы. Как показывает статистика, типичная программа на каждые 6–8 команд содержит 1 команду перехода. Последствия этого предсказать несложно: при наличии конвейера через каждые 6–8 команд его нужно очищать и заполнять заново в соответствии с адресом перехода. Все преимущества конвейеризации теряются. Поэтому в архитектуру Pentium был введен блок *предсказания переходов*. Суть этого метода заключается в следующем. Pentium имеет буфер адресов перехода, который хранит информацию о последних 256 переходах. Если некоторая команда управляет ветвлением, то в буфере запоминаются эта команда, адрес перехода и предположение о том, какая ветвь программы будет выполнена следующей. Почти в любой программе имеются циклы, в ходе выполнения которых периодически необходимо принимать решение либо о выходе из цикла, либо о переходе на его начало. Специальный блок предсказания адреса перехода прогнозирует, какое решение будет принято программой. При этом он основывается на предположении, что ветвь, которая была пройдена, будет использоваться снова, и загружает соответствующую команду перехода на конвейер. В случае если это предсказание верно, переход осуществляется без задержки. Для того чтобы судить об эффективности этого нововведения, достаточно отметить, что вероятность правильного предсказания составляет около 80 %.

Рассмотрим, как эти и некоторые другие свойства реализованы в структурной схеме последнего семейства микропроцессоров фирмы Intel P6 (Pentium Pro/II/III). Для иллюстрации используем рис. 1.2 (см. урок 1) и рис. 2.1.

На рис. 1.2 хорошо видно разделение кэш-памяти на две части — для кода и для данных. Это обеспечивает бесперебойную поставку машинных инструкций и элементов данных на конвейер микропроцессора. Исходные данные для кэш-памяти первого уровня обеспечивает кэш-память второго уровня. Заметьте, что информация из нее поступает на устройство шинного интерфейса и далее в соответствующую кэш-память первого уровня по 64-битнойшине. При этом благодаря более быстрому обновлению содержимого кэш-памяти первого уровня обеспечивается высокий темп работы микропроцессора.

Наиболее ценным свойством микропроцессорной архитектуры семейства Р6 является реализация механизма интеллектуальной обработки потока команд, называемого «динамическим выполнением». Этот механизм основывается на следующих свойствах, некоторые из них уже существовали сами по себе в прежних моделях микропроцессоров. Перечислим их:

О *Предсказание переходов, в том числе вложенных.* Эта технология не нова, однако конкретные модели микропроцессоров могут иметь некоторые особенност

сти ее реализации. В микропроцессорах ряда Р6 такая технология реализуется устройством выборки/декодирования (см. рис. 1.2). Основная задача механизма предсказания – исключить перезагрузку конвейера.

- *Динамический анализ потока данных.* Анализ проводится с целью определения зависимостей команд программы от данных и регистров процессора с последующей оптимизацией выполнения потока команд. Главный критерий здесь – максимально полная загрузка конвейера. Требование соблюдения данного критерия позволяет даже нарушать исходный порядок следования команд при поступлении на конвейер. Сбоя при этом не будет, так как внешне логика работы программы будет сохранена. Подобная внутренняя неупорядоченность исполнения команд позволяет держать конвейер загруженным даже в то время, когда данные в кэш-памяти второго уровня отсутствуют и необходимо тратить время на обращение за ними в оперативную память.
- *Интеллектуальное исполнение.* Это свойство характеризует способность микропроцессора реализовать неупорядоченное исполнение команд, восстановив впоследствии исходный порядок команд и организовав передачу результатов работы команд в порядке, предусмотренном исходным алгоритмом. Данная возможность обеспечивается разделением устройства выборки и исполнения команд и устройства формирования результата (см. рис. 1.2). Все промежуточные результаты работы команд во время их исполнения (нахождения их на конвейере) размещаются во временных регистрах. Блок удаления и восстановления постоянно просматривает буфер команд и ищет те из них, которые уже исполнены и не имеют связи по данным с другими командами или не находятся в ветвях незавершенных переходов. Когда такие команды найдены, устройство удаления и восстановления результатов помещает сформированные ими данные в память или регистры процессора в порядке, заданном исходным алгоритмом. После этого команды удаляются из конвейера.

Таким образом, реализация динамического исполнения команд позволяет организовать наиболее оптимальное прохождение команд программы через исполнительное устройство микропроцессора. А если учесть то, что в микропроцессорах семейства Р6 команды исполняются в три потока одновременно, то становятся понятными все преимущества такого подхода. Выше мы уже отмечали то, что конвейер микропроцессоров семейства Р6 имеет принципиальное отличие от конвейеров i486 и Pentium. Перечисленные выше три концепции представляют собой основу работы этого конвейера. Давайте еще конкретизируем схему микропроцессора, показанную на рис. 1.2. Это нам пригодится при обсуждении материала других уроков. Заметьте, что мы постепенно углубляемся в недра микропроцессора с тем, чтобы показать место языка ассемблера в его архитектуре и, как следствие этого рассуждения, продемонстрировать возможности языка. Расширенная схема микропроцессора ряда Р6 показана на рис. 2.1.

Строго говоря, на схеме рис. 2.1 показан только один из трех конвейеров микропроцессора и некоторые общие для всех трех конвейеров элементы (кэш-память, шины и т. д.). Схема рис. 2.1 представляет собой развитие рис. 1.2, на ней вы узнаете некоторые элементы, но они показаны более крупным планом. Из схемы рис. 2.1 видно, что структурно микропроцессор состоит из следующих подсистем.

Центральный процессор (Pentium Pro/II/III)

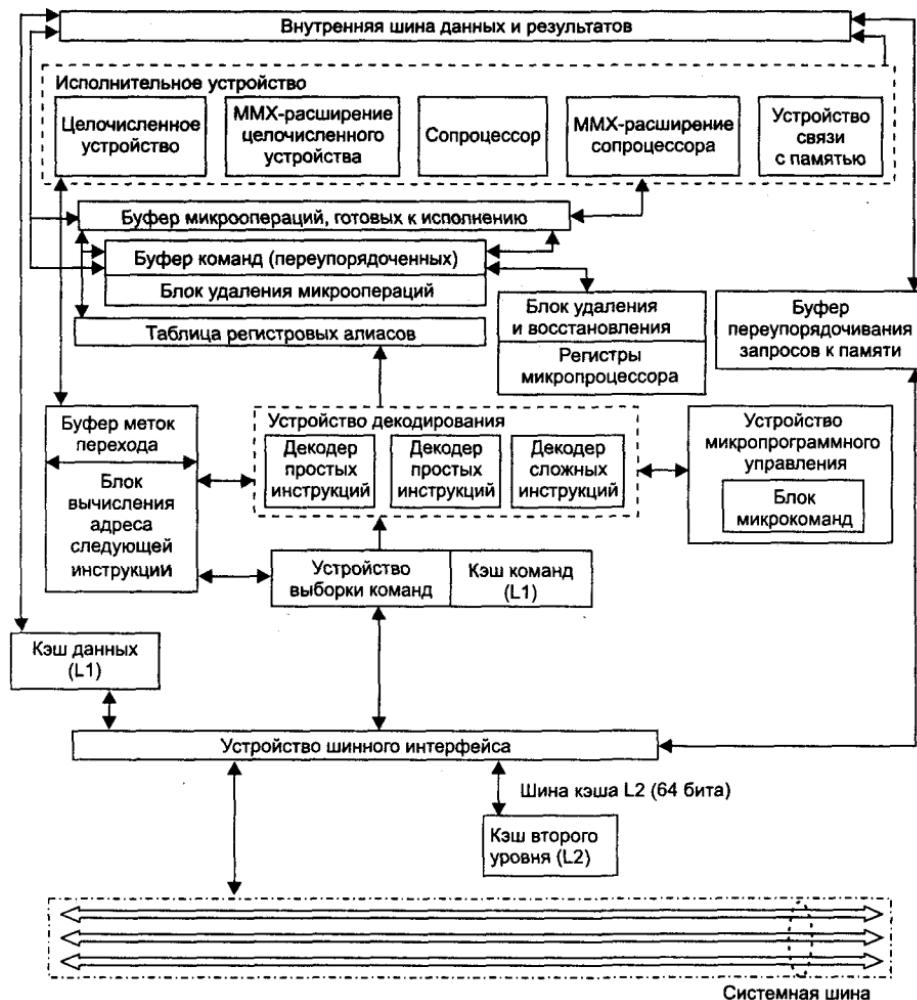


Рис. 2.1. Структурная схема микропроцессора семейства P6 (Pentium Pro/II/III)

- **Подсистема памяти.** Состоит из системной шины, кэша второго уровня L2, устройства шинного интерфейса, кэша первого уровня L1 (инструкций и данных), устройства связи с памятью и буфера переупорядочивания запросов к памяти.
- **Устройство выборки/декодирования.** Состоит из устройства выборки инструкций, буфера предсказаний переходов, декодера инструкций, блока микропрограммного управления и таблицы регистровых алиасов.
- **Буфер команд.** Содержит команды, переупорядоченные для оптимальной загрузки конвейера.
- **Устройство диспетчеризации/исполнения.** Содержит буфер микроопераций, готовых к исполнению, пять исполнительных устройств: два устройства для

исполнения целочисленных операций, два — с плавающей точкой и устройство связи с памятью. Мы допустили вольность в трактовании назначения исполнительных устройств, обозначив отдельные устройства для выполнения целочисленных, MMX-команд и MMX-команд с плавающей точкой. На самом деле такого деления нет. Сделано это было исключительно с учебной целью для того, чтобы перейти от рассмотрения архитектуры к рассмотрению системы команд ассемблера.

Опишем подробно порядок функционирования схемы рис. 2.1. Это описание не является строгим, кое-где, для лучшего понимания, оно упрощено. Ранее мы уже упомянули, что подсистема памяти для процессора семейства Р6 состоит из оперативной памяти, первичного (L1) и вторичного кэша (L2). Устройство шинного интерфейса обращается к оперативной памяти системы через внешнюю системную шину. Эта 64-разрядная шина ориентирована на обработку запросов, то есть каждый шинный запрос обрабатывается отдельно и требует обратной реакции. Пока устройство шинного интерфейса ожидает ответа на один запрос шины, возможно формирование многочисленных дополнительных запросов. Все они обслуживаются в порядке поступления. Считываемые по запросу данные помещаются в кэш второго уровня. То есть микропроцессор посредством устройства шинного интерфейса читает команды и данные из кэша второго уровня. Устройство шинного интерфейса взаимодействует с кэшем второго уровня через 64-разрядную шину кэша, которая также ориентирована на обработку запросов и работает на тактовой частоте процессора. Доступ к кэшу первого уровня осуществляется через внутренние шины на тактовой частоте микропроцессора. Синхронная работа с системной памятью кэш-памяти обоих уровней осуществляется благодаря специальному протоколу.

Запросы на операнды из памяти от команд в исполнительном устройстве микропроцессора обеспечиваются посредством *устройства связи с памятью* и *буфера переупорядочивания запросов к памяти*. Эти два устройства были специально включены в схему для того, чтобы обеспечить бесперебойное снабжение исполняемых команд необходимыми данными. Особо стоит подчеркнуть роль буфера переупорядочивания запросов к памяти. Он отслеживает все запросы к операндам в памяти и выполняет функции планирующего устройства. Если нужные для очередной операции данные в кэш-памяти данных (L1) отсутствуют, то буфер переупорядочивания запросов к памяти автоматически передает информацию о неудачном обращении к данным кэшу второго уровня (L2). Если и в кэше L2 нужных данных не оказалось, то буфер переупорядочивания запросов к памяти заставляет устройство шинного интерфейса сформировать запрос к оперативной памяти компьютера.

Устройство выборки/декодирования читает поток команд из кэша команд (L1) и декодирует их в последовательность микроопераций. Поток микроопераций (пока он еще соответствует последовательности исходных команд) поступает в буфер команд. *Устройство выборки* извлекает одну 32-байтную строку кэша команд за такт и передает ее в декодер. Устройство выборки вычисляет указатель на следующую команду, подлежащую выборке, на основании информации из таблицы меток перехода, состояния прерывания/исключения и сообщения от исполнительного целочисленного устройства об ошибке в предсказании метки перехода. Важная часть этого процесса — предсказание метки перехода, которое выполняется

по специальному алгоритму. В основе этого алгоритма лежит работа с *таблицей меток перехода*, которая содержит информацию о сделанных ранее переходах. Когда очередная команда, выбираемая из памяти, является командой перехода, то содержащийся в ней адрес перехода сравнивается с адресами, уже находящимися в таблице меток перехода. Если этого адреса нет в данной таблице, то выборка команд из памяти продолжается дальше до тех пор, пока не будет выполнена команда перехода исполнительным устройством. В результате ее выполнения будет подтверждена правильность перехода (в данном случае не перехода) — это будет в случае, если следующая команда соответствует условию (если оно есть) перехода. Если же этот адрес уже есть в таблице меток переходов, то на его основе устройство выборки формирует адрес следующей команды, подлежащей выборке. Аналогично, о правильности выборки этой команды будет известно после исполнения команды перехода исполнительным устройством. Если этот предсказанный переход был неверным, то конвейер будет сброшен и загружен заново в соответствии с адресом перехода. Цель правильного предсказания переходов в том, чтобы *устройство исполнения* постоянно было занято полезной работой, и сброс конвейера производился как можно реже.

Команды выбираются на конвейер устройством выборки команд, которое помещает их в *устройство декодирования*. Устройство декодирования состоит из трех параллельно работающих декодеров (два простых и один сложный). Декодеры преобразуют команды микропроцессора в микрооперации. Микрооперации представляют собой примитивные команды, которые выполняются пятью *исполнительными устройствами* микропроцессора, работающими параллельно. Многие машинные команды преобразуются в одиночные микрооперации (это делает простой декодер), а некоторые машинные команды преобразуются в последовательность от двух и более (оптимально — четырех) микроопераций (это делает сложный декодер). Информация о последовательности микроопераций для реализации конкретной машинной команды содержится в *блоке микропрограммного управления*. Кроме команд, декодеры обрабатывают также префиксы команд. Декодер команд может формировать до шести микроопераций за такт — по одной от простых декодеров и до четырех от сложного декодера. Для достижения наибольшей производительности работы декодеров необходимо, чтобы на их вход поступали команды, которые декодируются шестью микрооперациями в последовательности $4 + 1 + 1$. Если время работы программы критично, то имеет смысл провести ее оптимизацию, содержание которой заключается в переупорядочивании исходного набора команд таким образом, чтобы группы команд формировали последовательности микроопераций по схеме $4 + 1 + 1$. Количество микроопераций в той или иной команде можно найти в книге В. Юрова «Assembler: справочник»¹. Но помните, что проводить подобную оптимизацию есть смысл только для микропроцессоров семейства Р6 (Pentium Pro/II/III). После того как команды разбиты на микрооперации, порядок их выполнения трудно предсказать. При этом могут возникнуть проблемы с таким критичным ресурсом, как регистры. Суть здесь в том, что если в двух соседних фрагментах программы данные помещались в одинаковые регистры, откуда они, возможно, записывались в некоторые области памяти, а после переупорядочивания эти фрагменты перемешались, то как разобраться в том, какие регистры и где использовались. Эта проблема носит называ-

¹ Далее все ссылки на эту книгу будут просто ссылками на Справочник.

ние проблемы ложных взаимозависимостей и решается использованием механизма переименования регистров. Основу этого механизма составляет набор из 40 внутренних универсальных регистров, которые и используются в реальных вычислениях исполнительным устройством. Работа с этими регистрами абсолютно прозрачна для программ. Универсальные регистры могут работать как с целыми числами, так и со значениями с плавающей запятой. Информация о действительных именах регистров процессора и их внутренних именах (номерах универсальных регистров) помещается в *таблицу регистровых алиасов*.

В заключение процесса декодирования *устройство управления таблицей регистровых алиасов* добавляет к микрооперациям биты состояния и флаги, чтобы подготовить их к неупорядоченному выполнению, после чего посыпает получившиеся микрооперации в *буфер переупорядоченных команд*. Нужно заметить, что теперь порядок их следования не соответствует порядку следования соответствующих команд в исходной программе. Буфер переупорядоченных команд представляет собой массив ассоциативной памяти, физически выполненный в виде 40 регистров. Массив ассоциативной памяти представляет собой кольцевую структуру, элементы которой содержат два типа микроопераций: ожидающие своей очереди на исполнение и уже частично выполненные, но, из-за имевшего место переупорядочивания исполнения команд, не до конца. *Устройство диспетчеризации/исполнения* может выбирать микрооперации из этого буфера в любом порядке.

Устройство диспетчеризации/исполнения планирует и исполняет неупорядоченную последовательность микроопераций из буфера переупорядоченных команд. Но оно не занимается непосредственной выборкой микроопераций из буфера переупорядоченных команд, так как в нем могут содержаться и не готовые к исполнению микрооперации. Этим занимается устройство, управляющее специальным буфером, который условно назовем *буфером команд, готовых к исполнению*. Оно постоянно сканирует буфер переупорядоченных команд в поисках микроопераций, готовых к исполнению (фактически это означает, что все операнды для этих микроопераций доступны), после чего посыпает их соответствующим исполнительным устройствам, если они не заняты. Результаты исполнения микроопераций возвращаются в буфер переупорядоченных команд и сохраняются там наряду с другими микрооперациями до тех пор, пока не будут удалены *устройством удаления и восстановления*.

Подобная схема планирования и исполнения программ реализует классический принцип неупорядоченного выполнения, при котором микрооперации посыпаются исполнительным устройствам вне зависимости от их расположения в исходном алгоритме. В случае если к выполнению одновременно готовы две или более микрооперации одного типа (например целочисленные), то они выполняются в соответствии с принципом FIFO (First In, First Out — первым пришел, первым ушел), то есть в порядке поступления в буфер переупорядоченных команд.

Напомним, что исполнительное устройство состоит из пяти блоков, каждый из которых исполняет свой тип микроопераций: два целочисленных устройства, два устройства для вычислений с плавающей точкой и одно устройство связи с памятью. Таким образом, за один машинный такт одновременно исполняется пять микроопераций.

Два целочисленных исполнительных устройства могут параллельно обрабатывать две целочисленные микрооперации. Одно из этих целочисленных исполнитель-

ных устройств специально предназначено для работы с микрооперациями переходов. Оно способно обнаружить непредсказанный переход и сообщить об этом устройству выборки команд, чтобы перезапустить конвейер. Такая операция реализована следующим образом. Декодер команд отмечает каждую микрооперацию перехода и адрес перехода. Когда целочисленное исполнительное устройство выполняет микрооперацию перехода, то оно определяет, был ли предсказан переход или нет. Если переход предсказан правильно, то микрооперация отмечается пригодной для использования, и выполнение продолжается по предсказанный ветви. Если переход предсказан неправильно, то целочисленное исполнительное устройство изменяет состояние всех последующих микроопераций с тем, чтобы удалить их из буфера переупорядоченных команд. После этого целочисленное устройство помещает метку перехода в буфер меток перехода, который, в свою очередь, совместно с устройством выборки команд перезапускает конвейер относительно нового исполнительного адреса.

Устройство связи с памятью управляет загрузкой и сохранением данных для микроопераций. Для их загрузки в исполнительное устройство достаточно определить только адрес памяти, поэтому такое действие кодируется одной микрооперацией. Для сохранения данных необходимо определять и адрес, и записываемые данные, поэтому это действие кодируется двумя микрооперациями. Часть устройства связи с памятью, которое управляет сохранением данных, имеет два блока, позволяющие ему обработать адрес и данные для микрооперации параллельно. Это позволяет устройству связи с памятью выполнять загрузку и сохранение данных для микроопераций параллельно в одном тактовом цикле.

Исполнительные устройства с плавающей запятой аналогичны тем, что существуют в более ранних моделях микропроцессора Pentium. Было добавлено только несколько новых команд с плавающей запятой для организации условных переходов и перемещений.

Последний блок в этой схеме выполнения команд исходной программы — блок *удаления и восстановления*, задачей которого является возврат вычислительного процесса в рамки, определенные исходной последовательностью команд. Для этого он постоянно сканирует буфер переупорядоченных команд на предмет обнаружения полностью выполненных микроопераций, не имеющих связи с другими микрооперациями. Такие микрооперации удаляются из буфера переупорядоченных команд, восстанавливаются в порядке, соответствующем порядку команд исходной программы с учетом прерываний, исключений, точек прерывания и переходов. Блок удаления и восстановления может удалять три микрооперации за один машинный такт. При восстановлении команд в порядок, соответствующий исходному, блок удаления и восстановления записывает результаты в реальные регистры микропроцессора и в оперативную память.

На этом, наверное, следует завершить обсуждение общих вопросов, связанных с архитектурой микропроцессоров Intel. Им можно посвятить не одну увлекательную книгу, но это не является нашей целью. Наша ближайшая задача состоит в том, чтобы разобраться, как управлять этими сложнейшими микропроцессорами. В начале урока мы уже упоминали, что для этого нужно разобраться как с общей программной моделью компьютера вообще, так и с программной моделью микропроцессора в частности. В этих моделях описываются основные особенности

архитектуры компьютера, знание которых позволяет программисту эффективно и в полном объеме использовать все его возможности.

Любая выполняющаяся программа получает в свое распоряжение определенный набор ресурсов микропроцессора. Эти ресурсы необходимы для выполнения и хранения в памяти команд программы, данных и информации о текущем состоянии программы и микропроцессора. Набор этих ресурсов представляет собой *программную модель микропроцессора*. Схема, представленная на рис. 2.2, полностью соответствует программной модели микропроцессора Pentium III.

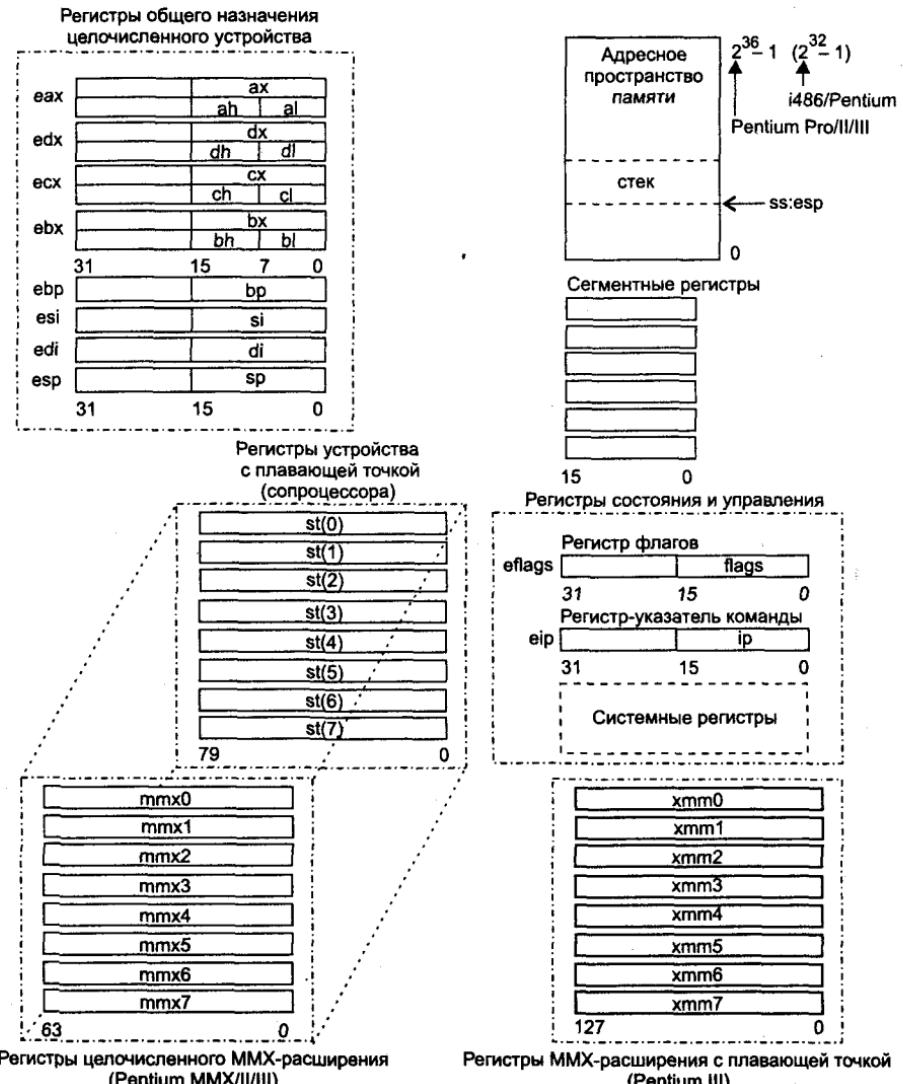


Рис. 2.2. Программная модель микропроцессора Intel (Pentium III)

Программные модели более ранних микропроцессоров (i486, Pentium) отличаются меньшим размером адресуемого пространства оперативной памяти (2^{32} - 1, так

как разрядность их шины адреса составляет 32 бита) и отсутствием некоторых групп регистров. Для каждой группы регистров в скобках обозначено, начиная с какой модели данной группа регистров появилась в программной модели микропроцессоров Intel. Если такого обозначения нет, то это означает, что данная группа регистров присутствовала в микропроцессорах i386 и i486. Более ранние микропроцессоры архитектуры Intel мы не рассматриваем ввиду их архаичности, но это вовсе не означает того, что данная книга не может использоваться для их программирования.

Итак, программную модель микропроцессора Intel составляют:

- пространство адресуемой памяти (для Pentium III – до 2^{36} – 1 байт);
- набор регистров для хранения данных общего назначения;
- набор сегментных регистров;
- набор регистров состояния и управления;
- набор регистров устройства вычислений с плавающей точкой (сопроцессора);
- набор регистров целочисленного MMX-расширения, отображенных на регистры сопроцессора (впервые появились в архитектуре микропроцессора Pentium MMX);
- набор регистров MMX-расширения с плавающей точкой (впервые появились в архитектуре микропроцессора Pentium III);
- программный стек. Это специальная информационная структура, работа с которой предусмотрена на уровне машинных команд. Более подробно мы будем обсуждать ее позже (и не один раз).

Как мы уже отметили ранее, программные модели ранних микропроцессоров Intel мы не рассматриваем, так как на самом деле они полностью представлены на схеме рис. 2.2 и составляют лишь небольшую ее часть. Так, в программную модель микропроцессора i8086 входят 8- и 16-битные регистры общего назначения, сегментные регистры, регистры flags, ip и адресное пространство памяти размером до 1 Мбайт.

Теперь рассмотрим основные компоненты программной модели микропроцессора. Начнем с обсуждения регистров.

Набор регистров

В программах на языке ассемблера регистры используются очень интенсивно. Большинство из них имеет определенное функциональное назначение. Как показано выше, программная модель микропроцессора имеет несколько групп регистров, доступных для использования в программах. С точки зрения программиста, их можно разделить на две части:

- пользовательские регистры, к которым относятся:
 - регистры общего назначения eax/ax/ah/a1, ebx/bx/bh/b1, edx/dx/dh/d1, ecx/cx/ch/c1, ebp/bp, esi/si, edi/di, esp/sp. Регистры этой группы используются для хранения данных и адресов;

- сегментные регистры `cs`, `ds`, `ss`, `es`, `fs`, `gs`. Регистры этой группы используются для хранения адресов сегментов в памяти;
- регистры сопроцессора `st(0)`, `st(1)`, `st(2)`, `st(3)`, `st(4)`, `st(5)`, `st(6)`, `st(7)`. Регистры этой группы предназначены для написания программ, использующих тип данных с плавающей точкой (см. урок 19);
- целочисленные регистры MMX-расширения `mmx0`, `mmx1`, `mmx2`, `mmx3`, `mmx4`, `mmx5`, `mmx6`, `mmx7` (см. урок 20);
- регистры MMX-расширения с плавающей точкой `xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, `xmm7` (см. урок 20);
- регистры состояния и управления — это регистры, которые содержат информацию о состоянии микропроцессора, исполняемой программы и позволяют изменить это состояние:
- регистр флагов `eflags/flags`;
- регистр указатель команды `eip/ip`;
- системные регистры — это регистры для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели микропроцессора (более подробно рассмотрены на уроках 16 и 17). На схеме рис. 2.2 регистры этой группы не показаны по двум причинам: во-первых, их достаточно много, и, во-вторых, состав их может отличаться для различных моделей микропроцессора.

Почему многие из этих регистров приведены с наклонной разделительной чертой? Нет, это не разные регистры — это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты. Зачем так сделано? Для обеспечения работоспособности программ, написанных для младших 16-разрядных моделей микропроцессоров фирмы Intel, начиная с i8086. Микропроцессоры i486 и Pentium имеют, в основном, 32-разрядные регистры. Их количество, за исключением сегментных регистров, такое же, как и у i8086, но размерность больше, что и отражено в их обозначениях, — они имеют приставку `e` (Extended).

Многие из приведенных регистров предназначены для работы с определенными вычислительными подсистемами микропроцессора: сопроцессором, MMX-расширениями. Поэтому их целесообразно рассматривать в соответствующем контексте. Так как первая часть книги посвящена вопросам программирования целочисленной подсистемы микропроцессора, то мы и начнем обсуждение с регистров, обеспечивающих ее функционирование. В дальнейшем при обсуждении новых вычислительных подсистем микропроцессора мы будем рассматривать и соответствующие регистры.

Регистры общего назначения

Регистры общего назначения используются в программах для хранения:

- операндов логических и арифметических операций;
- компонентов адреса;
- указателей на ячейки памяти.

В принципе, все эти регистры доступны для хранения операндов без особых ограничений, хотя при определенных условиях некоторые из них все же имеют жесткое функциональное назначение, закрепленное на уровне логики работы машинных команд. Среди всех этих регистров особо следует выделить регистр esp. Его не следует использовать явно для хранения каких-либо операндов программы, так как в нем хранится указатель на положение вершины стека программы.

Все регистры этой группы позволяют обращаться к своим «младшим» частям (рис. 2.2). Рассматривая этот рисунок, заметьте, что использовать для самостоятельной адресации можно только младшие 16- и 8-битные части этих регистров. Старшие 16 битов этих регистров как самостоятельные объекты недоступны. Это сделано, как мы отметили выше, для совместимости с младшими 16-разрядными моделями микропроцессоров фирмы Intel. Перечислим регистры, относящиеся к группе регистров общего назначения. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их еще называют *регистрами АЛУ*:

- eax/ax/ah/a1 (Accumulator register) – *аккумулятор*. Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;
- ebx/bx/bh/b1 (Base register) – *базовый регистр*. Применяется для хранения базового адреса некоторого объекта в памяти;
- ecx/cx/ch/c1 (Count register) – *регистр-счетчик*. Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла `loop`, кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра ecx/cx;
- edx/dx/dh/d1 (Data register) – *регистр данных*. Так же как и регистр eax/ax/ah/a1, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

Следующие два регистра используются для поддержки так называемых цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

- esi/si (Source Index register) – *индекс источника*. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;
- edi/di (Destination Index register) – *индекс приемника (получателя)*. Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как *стек*. В свое время мы подробно познакомимся с тем, как его использовать. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

- esp/sp (Stack Pointer register) – *регистр указателя стека*. Содержит указатель вершины стека в текущем сегменте стека;
- ebp/bp (Base Pointer register) – *регистр указателя базы кадра стека*. Предназначен для организации произвольного доступа к данным внутри стека.

Не спешите пугаться столь жесткого функционального назначения регистров АЛУ. На самом деле большинство из них может использоваться при программировании для хранения операндов практически в любых сочетаниях. Возможные варианты использования этих регистров приведены в синтаксических диаграммах команд в Справочнике. Но, как мы отметили выше, некоторые команды используют фиксированные регистры для выполнения своих действий. Это нужно обязательно учитывать. Использование жесткого закрепления регистров для некоторых команд позволяет более компактно кодировать их машинное представление. Знание этих особенностей позволит вам при необходимости хотя бы на несколько байт сэкономить память, занимаемую кодом программы.

Сегментные регистры

В программной модели микропроцессора имеется шесть *сегментных регистров*: cs, ss, ds, es, gs, fs. Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых *сегментами*. Соответственно, такая организация памяти называется *сегментной*. Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены сегментные регистры. Фактически, с небольшой поправкой, как мы увидим далее, в этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

1. *Сегмент кода*. Содержит команды программы. Для доступа к этому сегменту служит регистр cs (code segment register) — *сегментный регистр кода*. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор (то есть эти команды загружаются в конвейер микропроцессора);
2. *Сегмент данных*. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр ds (data segment register) — *сегментный регистр данных*, который хранит адрес сегмента данных текущей программы;
3. *Сегмент стека*. Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по следующему принципу: последний записанный в эту область элемент выбирается первым. Для доступа к этому сегменту служит регистр ss (stack segment register) — *сегментный регистр стека*, содержащий адрес сегмента стека;
4. *Дополнительный сегмент данных*. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их

адреса требуется указывать явно с помощью специальных префиксов перед определения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

Регистры состояния и управления

В микропроцессор включены несколько регистров (см. рис. 2.2), которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер.

К этим регистрам относятся:

- регистр флагов eflags flags;
- регистр указателя команды eip/ip.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. Рассмотрим подробнее назначение и содержимое этих регистров:

eflags/flags (flag register) — *регистр флагов*. Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются *флагами*. Младшая часть этого регистра полностью аналогична регистру flags для i8086. На рис. 2.3 показано содержимое регистра eflags.

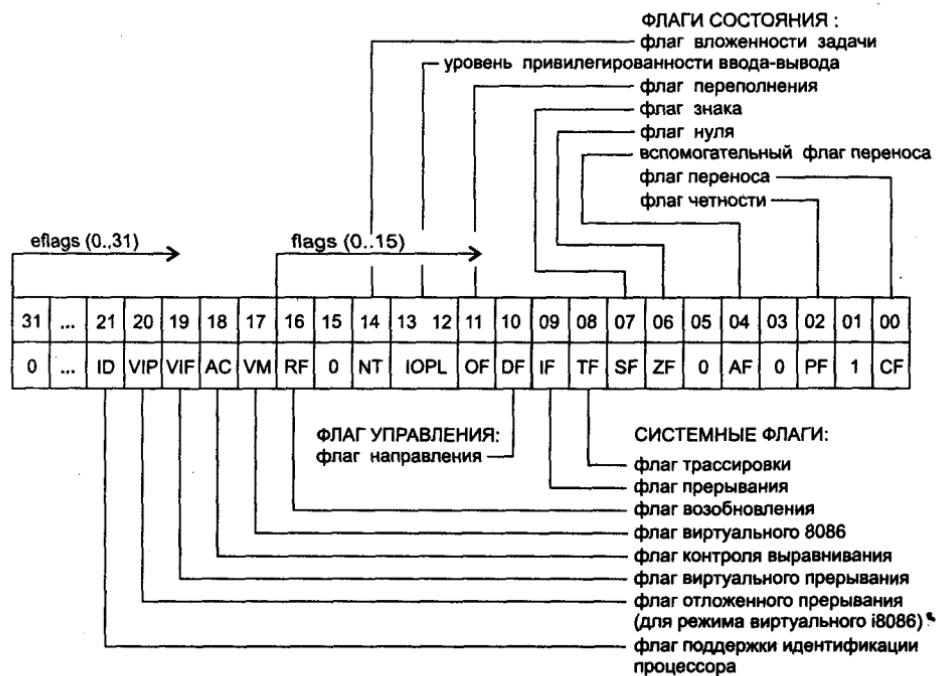


Рис. 2.3. Содержимое регистра eflags

Исходя из особенностей использования, флаги регистра `eflags/flags` можно разделить на три группы.

- **8 флагов состояния.** Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра `eflags` отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм. В табл. 2.1 приведены флаги состояния и указано их назначение;
- **1 флаг управления.** Обозначается как `df` (Directory Flag). Он находится в десятом бите регистра `eflags` и используется цепочечными командами. Значение флага `df` определяет направление поэлементной обработки в этих операциях: от начала строки к концу (`df = 0`) либо, наоборот, от конца строки к ее началу (`df = 1`). Для работы с флагом `df` существуют специальные команды `cld` (снять флаг `df`) и `std` (установить флаг `df`). Применение этих команд позволяет привести флаг `df` в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;
- **5 системных флагов,** управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы. В табл. 2.2 перечислены системные флаги и их назначение.

`eip/ip` (Instruction Pointer register) – *указатель команд*. Регистр `eip/ip` имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра `cs` в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра `eip/ip`.

Таблица 2.1. Флаги состояния

| Мнемоника флага | Флаг | Номер бита в <code>eflags</code> | Содержание и назначение |
|-----------------|-----------------------------|----------------------------------|---|
| <code>cf</code> | Флаг переноса (Carry Flag) | 0 | 1 – арифметическая операция произвела перенос из старшего бита результата. Старшим является 7-й, 15-й или 31-й бит в зависимости от размерности операнда; 0 – переноса не было |
| <code>pf</code> | Флаг паритета (Parity Flag) | 2 | 1 – 8 младших разрядов (этот флаг – только для 8 младших разрядов операнда любого размера) результата содержат четное число единиц; 0 – 8 младших разрядов результата содержат нечетное число единиц |

Таблица 2.1 (продолжение)

| Мнемоника флага | Флаг | Номер бита в eflags | Содержание и назначение |
|-----------------|--|---------------------|--|
| af | Вспомогательный флаг переноса (Auxiliary carry Flag) | 4 | Только для команд, работающих с BCD-числами. Фиксирует факт заема из младшей тетрады результата: 1 – в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде; 0 – переносов и заемов в (из) 3 разряд(а) младшей тетрады результата не было |
| zf | Флаг нуля (Zero Flag) | 6 | 1 – результат нулевой; 0 – результат ненулевой |
| sf | Флаг знака (Sign Flag) | 7 | Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов, соответственно): 1 – старший бит результата равен 1; 0 – старший бит результата равен 0 |
| of | Флаг переполнения (Overflow Flag) | 11 | Флаг of используется для фиксирования факта потери значащего бита при арифметических операциях: 1 – в результате операции происходит перенос (заем) в (из) старшего, знакового бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов, соответственно); 0 – в результате операции не происходит переноса (заема) в (из) старшего, знакового бита результата |
| iopl | Уровень привилегий ввода-вывода (Input/Output Privilege Level) | 12, 13 | Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода, в зависимости от привилегированности задачи |
| nt | Флаг вложенности задачи (Nested Task) | 14 | Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую |

Таблица 2.2. Системные флаги

| Мнемоника флага | Флаг | Номер бита в eflags | Содержание и назначение |
|-----------------|---|---------------------|--|
| tf | Флаг трассировки (Trace Flag) | 8 | Предназначен для организации пошаговой работы микропроцессора: 1 – микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; 0 – обычная работа |
| if | Флаг прерывания (Interrupt enable Flag) | 9 | Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR): 1 – аппаратные прерывания разрешены; 0 – аппаратные прерывания запрещены |
| rf | Флаг возобновления (Resume Flag) | 16 | Используется при обработке прерываний от регистров отладки |
| vm | Флаг виртуального 8086 (Virtual 8086 Mode) | 17 | Признак работы микропроцессора в режиме виртуального 8086: 1 – процессор работает в режиме виртуального 8086; 0 – процессор работает в реальном или защищенном режиме |
| ac | Флаг контроля выравнивания (Alignment Check) | 18 | Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом am в системном регистре cr0. К примеру, Pentium разрешает размещать команды и данные с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам, кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут возбуждать исключительную ситуацию |
| vif | Флаг виртуального прерывания (Virtual Interrupt Flag) | 19 | При определенных условиях (одно из которых – работа микропроцессора в V-режиме) является аналогом флага if. Флаг vif используется совместно с флагом vip. Флаг появился в микропроцессоре Pentium |
| vip | Флаг отложенного виртуального прерывания (Virtual Interrupt Pending flag) | 20 | Устанавливается в 1 для индикации отложенного прерывания. Используется при работе в V-режиме совместно с флагом vif. Флаг появился в микропроцессоре Pentium |

Таблица 2.2 (продолжение)

| Мнемоника флага | Флаг | Номер бита в eflags | Содержание и назначение |
|-----------------|--|---------------------|---|
| id | Флаг идентификации (IDentification flag) | 21 | Используется для того, чтобы показать факт поддержки микропроцессором инструкции <code>cpuid</code> . Если программа может установить или очистить этот флаг, то это означает, что данная модель микропроцессора поддерживает инструкцию <code>cpuid</code> |

Организация памяти

Физическая память, к которой микропроцессор имеет доступ по шине адреса (см. рис. 1.2), называется *оперативной памятью* (или оперативным запоминающим устройством – ОЗУ). На самом нижнем уровне память компьютера можно рассматривать как массив битов. Один бит может хранить значение 0 или 1. Для физической реализации битов и работы с ними идеально подходят логические схемы. Но микропроцессору неудобно работать с памятью на уровне битов, поэтому реально ОЗУ организовано как последовательность ячеек – *байтов*. Один байт состоит из 8 бит. Каждому байту соответствует свой уникальный адрес (его номер), называемый *физическими*. Диапазон значений физических адресов зависит от разрядности шины адреса микропроцессора. Для i486 и Pentium он находится в пределах от 0 до $2^{32} - 1$ (4 Гбайт). Для микропроцессоров семейства P6 (Pentium Pro/II/III) этот диапазон шире – от 0 до $2^{36} - 1$ (64 Гбайт).

Механизм управления памятью полностью аппаратный. Это означает, что программа не может сама сформировать физический адрес памяти на адреснойшине. Ей приходится «играть» по правилам микропроцессора. Что это за правила, мы узнаем чуть ниже. Пока же отметим, что в конечном итоге этот механизм позволяет обеспечить:

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

Микропроцессор аппаратно поддерживает две модели использования оперативной памяти:

- *сегментированную модель*. В этой модели программе выделяются непрерывные области памяти (сегменты), а сама программа может обращаться только к тем, которые находятся в этих сегментах;
- *страничную модель*. Ее можно рассматривать как надстройку над сегментированной моделью. В случае использования этой модели оперативная память рассматривается как совокупность блоков фиксированного размера (4 Кбайт). Основное применение этой модели связано с организацией виртуальной па-

мяти, что позволяет операционной системе использовать для работы программ пространство памяти большее, чем объем физической памяти. Для микропроцессоров i486 и Pentium размер возможной виртуальной памяти может достигать 4 Тбайт.

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

- *Режим реальных адресов*, или просто *реальный режим*. Это режим, в котором работал i8086. Наличие его в i486 и Pentium обусловлено тем, что фирма Intel старается обеспечить в новых моделях микропроцессоров функционирование программ, разработанных для ранних моделей микропроцессоров.
- *Защищенный режим*. Этот режим позволяет максимально реализовать все архитектурные идеи, заложенные в модели микропроцессоров Intel, начиная с i80286. Программы, разработанные для i8086 (реального режима), не могут функционировать в защищенном режиме. Одна из причин этого связана именно с особенностями формирования физического адреса в защищенном режиме.
- *Режим виртуального 8086*. Переход в этот режим возможен, если микропроцессор уже находится в защищенном режиме. Отличительной особенностью этого режима является возможность одновременной работы нескольких программ, разработанных для i8086. Несмотря на то что микропроцессор находился в защищенном режиме, в режиме виртуального i8086 возможна работа программ реального режима. Это объясняется тем, что процесс формирования физического адреса для этих программ производится по правилам реального режима.
- *Режим системного управления* — это новый режим работы микропроцессора, впервые появившийся в микропроцессоре Pentium. Он обеспечивает операционную систему механизмом для выполнения машинно-зависимых функций, таких как перевод компьютера в режим пониженного энергопотребления или выполнения действий по защите системы. Для перехода в данный режим микропроцессор должен получить специальный сигнал — SMI — от усовершенствованного программируемого контроллера прерываний APIC (Advanced Programmable Interrupt Controller), при этом сохраняется состояние вычислительной среды микропроцессора. Функционирование микропроцессора в этом режиме подобно его работе в режиме реальных адресов. Возврат из этого режима производится специальной командой микропроцессора.

В этой книге мы еще вернемся к подробному обсуждению различных аспектов реального и защищенного режимов. На данном уроке рассмотрим только особенности работы с оперативной памятью для реального режима, в котором поддерживается только сегментированная модель организации памяти. Все вопросы, связанные с защищенным режимом, и, в частности, особенности организации памяти в этом режиме рассмотрим на уроке 16.

Сегментированная модель памяти

Вначале постараемся формально определить фундаментальные понятия сегмента и сегментации.

Сегментация — механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния. В основе механизма сегментации лежит понятие *сегмента*, который представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти.

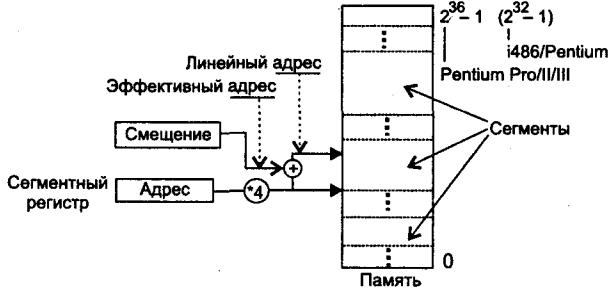
Когда мы рассматривали сегментные регистры, то отмечали, что для микропроцессоров Intel, начиная с i8086, принят особый подход к управлению памятью. Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет только к трем основным сегментам: кода, данных и стека, — и от одного до трех дополнительных сегментов данных. Программа никогда не знает, по каким физическим адресам будут размещены ее сегменты. Этим занимается операционная система. Операционная система размещает сегменты программы в оперативной памяти по определенным физическим адресам, после чего помещает значения этих адресов в определенные места. Куда именно, зависит от режима работы микропроцессора. Так, в реальном режиме эти адреса помещаются непосредственно в соответствующие сегментные регистры, а в защищенном режиме они размещаются в элементы специальной системной дескрипторной таблицы. Внутри сегмента программа обращается к адресам относительно начала сегмента линейно, то есть начиная с 0 и заканчивая адресом, равным размеру сегмента. Этот относительный адрес, или *смещение*, который микропроцессор использует для доступа к данным внутри сегмента, называется *эффективным*.

Отличия моделей сегментированной организации памяти в различных режимах хорошо видны на схеме (рис. 2.4). Различают три основных модели сегментированной организации памяти:

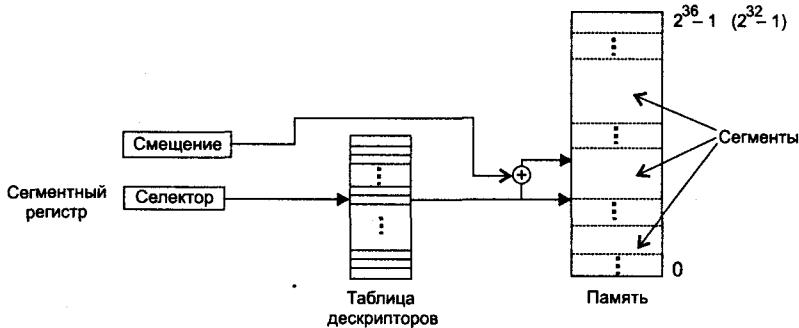
- сегментированная модель памяти реального режима;
- сегментированная модель памяти защищенного режима;
- сплошная модель памяти защищенного режима.

Рассмотрим порядок формирования физического адреса в реальном режиме. Порядок формирования физического адреса в защищенном режиме мы рассмотрим на уроке 16. Под *физическими адресами* понимается адрес памяти, выдаваемый на шину адреса микропроцессора (см. рис. 1.2). Другое название этого адреса — *линейный адрес*. Эта двойственность в названии обусловлена наличием страничной модели организации оперативной памяти. Эти названия являются синонимами только при отключении страничного преобразования адреса (в реальном режиме страничная адресация всегда отключена). Страницчная модель, как мы отметили выше, является надстройкой над сегментированной моделью. В страницной модели линейный адрес и физический адрес имеют разные значения. Далее мы будем обсуждать рис. 2.5, на котором показан порядок формирования адреса в реальном режиме работы микропроцессора. Обратите внимание на наличие в этой схеме устройства страничного преобразования адреса. Это устройство предназначено для того, чтобы совместить две принципиально разные модели организации оперативной памяти и выдать на шину адреса истинное значение физического адреса памяти.

Сегментированная модель памяти реального режима



Сегментированная модель памяти защищенного режима



Сплошная модель памяти защищенного режима

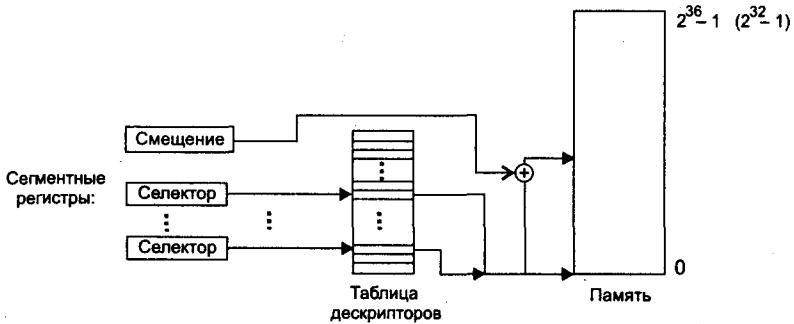


Рис. 2.4. Модели сегментированной организаций памяти микропроцессоров Intel

Формирование физического адреса в реальном режиме

В реальном режиме механизм адресации физической памяти имеет следующие характеристики:

- диапазон изменения физического адреса от 0 до 1 Мбайт. Эта величина определяется тем, что шина адреса i8086 имела 20 линий;
- максимальный размер сегмента 64 Кбайт. Это объясняется 16-разрядной архитектурой i8086. Нетрудно подсчитать, что максимальное значение, которое

могут содержать 16-разрядные регистры, составляет $2^{16} - 1$, что применительно к памяти и определяет величину 64 Кбайт;

- для обращения к конкретному физическому адресу оперативной памяти необходимо определить адрес начала сегмента (сегментную составляющую) и смещение внутри сегмента. Но мы помним, что сегментная составляющая адреса (или *база сегмента*) представляет собой всего лишь 16-битное значение, помещенное в один из сегментных регистров. Максимальное значение, которое при этом получается, соответствует $2^{16} - 1$. Если так рассуждать, то получается, что адрес начала сегмента может быть только в диапазоне 0–64 Кбайт от начала оперативной памяти. Возникает вопрос о том, как адресовать остальную часть оперативной памяти вплоть до 1 Мбайт с учетом того, что размер самого сегмента не превышает 64 Кбайт. Дело в том, что в сегментном регистре содержатся только старшие 16 бит физического адреса начала сегмента. Недостающие младшие четыре бита 20-битного адреса получаются сдвигом значения в сегментном регистре влево на 4 разряда. Эта операция сдвига выполняется аппаратно и для программного обеспечения абсолютно прозрачна. Получившееся 20-битное значение и является настоящим физическим адресом, соответствующим началу сегмента. Что касается второго компонента, участвующего в образовании физического адреса некоторого объекта в памяти – смещения, – то оно представляет собой 16-битное значение. Это значение может содержаться явно в команде либо косвенно в одном из регистров общего назначения. В микропроцессоре эти две составляющие складываются на аппаратном уровне, в результате чего получается физический адрес памяти размерностью 20 бит. Данный механизм образования физического адреса позволяет сделать программное обеспечение перемещаемым, то есть не зависящим от конкретных адресов загрузки его в оперативной памяти. Он показан на рис. 2.5.

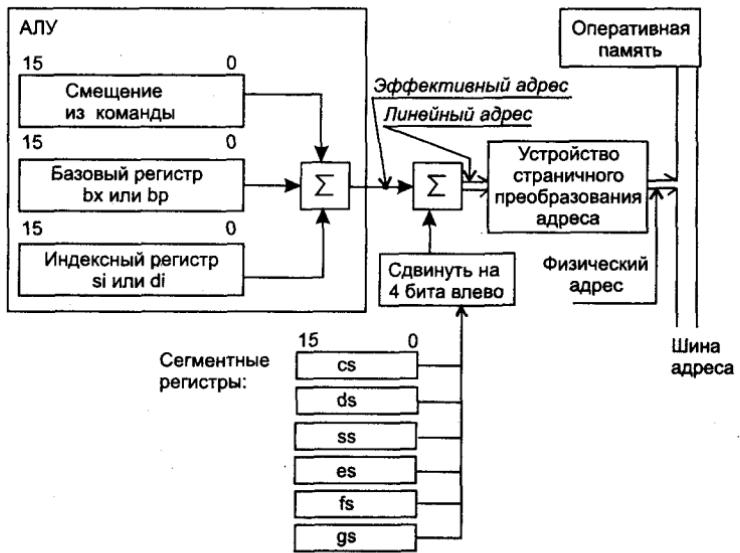


Рис. 2.5. Механизм формирования физического адреса в реальном режиме.

На рис. 2.5 хорошо видно, как формируется некоторый целевой физический адрес: сегментная часть извлекается из одного из сегментных регистров, сдвигается на четыре разряда влево и суммируется со смещением. В свою очередь, видно, что значение смещения можно получить минимум из одного и максимум из трех источников: из значения смещения в самой машинной команде и (или) из содержимого одного базового и (или) одного индексного регистра. Количество источников, участвующих в формировании смещения, определяется кодированием конкретной машинной команды, и если таких источников несколько, то значения в них складываются. В заключение заметим, что не стоит волноваться насчет того, что существует несоответствие размеров шины адреса микропроцессора i486 или Pentium (32 бита) и 20-битного значения физического адреса реального режима. Пока микропроцессор находится в реальном режиме, старшие 12 линий шины адреса попросту недоступны, хотя при определенных условиях и существует возможность работы с первыми 64 Кбайт оперативной памяти, лежащими сразу после первого мегабайта.

Недостатки такой организации памяти:

- сегменты бесконтрольно размещаются с любого адреса, кратного 16 (так как содержимое сегментного регистра аппаратно смещается на 4 разряда). Как следствие, программа может обращаться по любым адресам, в том числе и реально не существующим;
- сегменты имеют максимальный размер 64 Кбайт;
- сегменты могут перекрываться с другими сегментами.

Желанием ввести в архитектуру средства, позволяющие избавиться от указанных недостатков, в частности, и обусловлено появление защищенного режима (см. урок 16).

Типы данных

Понятие *типа данных* носит двойственный характер. С точки зрения размерности микропроцессор аппаратно поддерживает следующие основные типы данных (рис. 2.6).

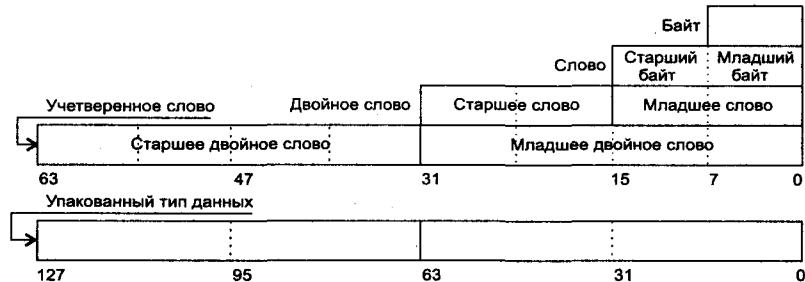


Рис. 2.6. Основные типы данных микропроцессора

- **Байт** — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом.

- **Слово** — последовательность из двух байтов, имеющих последовательные адреса. Размер слова — 16 битов; биты в слове нумеруются от 0 до 15. Байт, содержащий нулевой бит, называется **младшим байтом**, а байт, содержащий 15-й бит — **старшим байтом**. Микропроцессоры Intel имеют важную особенность — младший байт всегда хранится по меньшему адресу. Адресом слова считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.
- **Двойное слово** — последовательность из четырех байтов (32 бита), расположенных по последовательным адресам. Нумерация этих битов производится от 0 до 31. Слово, содержащее нулевой бит, называется **младшим словом**, а слово, содержащее 31-й бит, — **старшим словом**. Младшее слово хранится по меньшему адресу. Адресом двойного слова считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.
- **Учетверенное слово** — последовательность из восьми байт (64 бита), расположенных по последовательным адресам. Нумерация битов производится от 0 до 63. Двойное слово, содержащее нулевой бит, называется **младшим двойным словом**, а двойное слово, содержащее 63-й бит, — **старшим двойным словом**. Младшее двойное слово хранится по меньшему адресу. Адресом учетверенного слова считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.
- **128-битный упакованный тип данных**. Этот тип данных появился в микропроцессоре Pentium III. Для работы с ним в микропроцессор введены специальные команды (см. урок 20).

Кроме трактовки типов данных с точки зрения их разрядности, микропроцессор на уровне команд поддерживает логическую интерпретацию этих типов (рис. 2.7).

- **Целый тип со знаком** — двоичное значение со знаком размером 8, 16 или 32 бита. Знак в этом двоичном числе содержится в 7, 15 или 31 бите соответственно. Ноль в этих битах в операндах соответствует положительному числу, а единица — отрицательному. Отрицательные числа представляются в дополнительном коде. Числовые диапазоны для этого типа данных следующие:

8-разрядное целое — от -128 до +127;

16-разрядное целое — от -32 768 до +32 767;

32-разрядное целое — от -2^{31} до $+2^{31} - 1$.

- **Целый тип без знака** — двоичное значение *без знака*, размером 8, 16 или 32 бита. Числовой диапазон для этого типа следующий:

байт — от 0 до 255;

слово — от 0 до 65 535;

двойное слово — от 0 до $2^{32} - 1$.

- **Указатель на память** бывает двух типов.

- **Ближний тип** — 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента. Эти указатели могут

также использоваться в сплошной (плоской) модели памяти, где сегментные составляющие одинаковы.

- **Дальний тип** — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — *селектора* и 32-разрядного смещения.

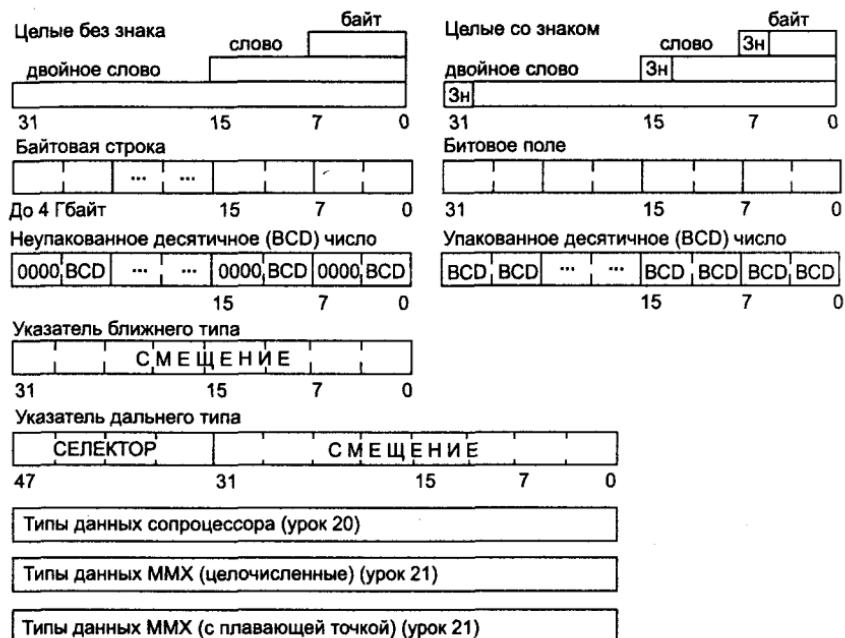


Рис. 2.7. Основные логические типы данных микропроцессора

- **Цепочка** представляет собой некоторый непрерывный набор байтов, слов или двойных слов максимальной длиной до 4 Гбайт.
- **Битовое поле** представляет собой непрерывную последовательность битов, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита любого байта и содержать до 32 битов.
- **Неупакованный двоично-десятичный тип** — байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом.
- **Упакованный двоично-десятичный тип** представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4–7) является старшей.
- **Типы данных с плавающей точкой.** Сопроцессор имеет несколько собственных типов данных, несовместимых с типами данных целочисленного устройства. Подробно этот вопрос обсуждается на уроке 19.

- *Типы данных MMX-расширения (Pentium MMX/II).* Данный тип данных появился в микропроцессоре Pentium MMX. Он представляет собой совокупность упакованных целочисленных элементов определенного размера. Подробно этот вопрос обсуждается на уроке 20.
- *Типы данных MMX-расширения (Pentium III).* Этот тип данных появился в микропроцессоре Pentium III. Он представляет собой совокупность упакованных элементов с плавающей точкой фиксированного размера. Подробно этот вопрос обсуждается на уроке 20.

Отметим, что «Зн» на рис. 2.7 означает знаковый бит.

Формат команд

Программирование на уровне машинных команд — это тот минимальный уровень, на котором еще возможно программирование компьютера. Система машинных команд должна быть достаточной для того, чтобы реализовать требуемые действия, выдавая указания аппаратуре машины. Каждая машинная команда состоит из двух частей: *операционной части*, определяющей, «что делать», и *операндной части*, определяющей объекты обработки, то есть то, «над чем делать».

Вся эта информация, разумеется, должна быть определенным образом закодирована и формализована. Мы вернемся к этому вопросу позже на уроке 6, где подробно разберемся со структурой и правилами формирования машинной команды. Отметим только интересующий нас архитектурный аспект.

В машинную команду микропроцессора явно или неявно входят следующие элементы:

- *поле префиксов* — элемент команды, который уточняет либо модифицирует действие этой команды в следующих аспектах:
 - замена сегмента, если нас по какой-либо причине не удовлетворяет сегмент по умолчанию;
 - изменение размерности адреса;
 - изменение размерности операнда;
 - указание на необходимость повторения данной команды;
- *поле кода операции*, определяющее действие данной команды. Одной и той же команде могут соответствовать несколько кодов операций, в зависимости от ее operandов;
- *поле operandов*; содержит от 0 до 2 элементов.

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

По этой причине возможны только следующие сочетания operandов в команде:

- регистр — регистр;
- регистр — память;
- память — регистр;
- непосредственный операнд — регистр;
- непосредственный операнд — память.

Обработка прерываний

По определению *прерывание* означает временное прекращение основного процесса вычислений для выполнения некоторых запланированных или незапланированных действий, вызываемых работой аппаратуры или программы. Эти действия могут носить сервисный характер, быть запросами со стороны программы пользователя на выполнение обслуживания со стороны операционной системы либо быть реакцией на нештатные ситуации.

Механизм прерываний поддерживается на аппаратном уровне и позволяет реализовать как эффективное взаимодействие программ с операционной системой, так и эффективное управление программой аппаратной частью компьютера.

В зависимости от источника, прерывания классифицируются так:

- аппаратные*, возникающие как реакция микропроцессора на физический сигнал от некоторого устройства компьютера (клавиатура, системный таймер, жесткий диск и т. д.). По времени возникновения эти прерывания *асинхронны*, то есть происходят в случайные моменты времени;
- программные*, которые вызываются искусственно с помощью соответствующей команды из программы (команда `int`). Они предназначены для выполнения некоторых действий операционной системы. Эти прерывания являются *синхронными*;
- исключения* — разновидность программных прерываний, являющихся реакцией микропроцессора на нестандартную ситуацию, возникшую *внутри* микропроцессора во время выполнения некоторой команды программы.

Более подробно механизм прерываний будет обсуждаться на уроке 15.

Подведем некоторые итоги:

- Понимание архитектуры ЭВМ является ключевым для изучения ассемблера. Это касается любого типа компьютера. Структура ассемблера, формат его команд, адресация operandов и т. д. полностью отражают особенности архитектуры компьютера. Есть общие архитектурные свойства, присущие всем современным машинам фон-неймановской архитектуры, и есть частные свойства, присущие конкретному типу компьютеров.
- Целью изучения архитектуры является:
 - выявление набора доступных для программирования регистров, их функционального назначения и структуры;

- понимание организации оперативной памяти и порядка ее использования;
- знакомство с типами данных;
- изучение формата машинных команд;
- выяснение организации обработки прерываний.

- Микропроцессор содержит 32 доступных тем или иным образом регистра. Они делятся на пользовательские и системные регистры.
- Пользовательские регистры имеют определенное функциональное назначение. Среди них особо нужно выделить регистр флагов `eflags` и регистр указателя команды `eip`. Назначение регистра `eflags` — отражать состояние микропроцессора после выполнения последней машинной команды. Регистр `eip` содержит адрес следующей выполняемой машинной команды. Доступ к этим регистрам, в силу их специфики, со стороны программ пользователя ограничен.
- Микропроцессор имеет три режима работы:
- реальный режим, который использовался для i8086 и поддерживается до сих пор для обеспечения совместимости программного обеспечения;
 - защищенный режим, который впервые появился в i80286;
 - режим виртуального i8086. Обеспечивает полную эмуляцию микропроцессора i8086, позволяя при этом организовать многозадачную работу нескольких таких программ.
- Микропроцессор имеет сложную систему управления памятью, работа которой зависит от режима микропроцессора.
- Микропроцессор благодаря гибкой системе команд поддерживает довольно большую номенклатуру типов данных.



3

УРОК

Простая программа на ассемблере

-
- «Ассемблерный» уровень разработки программы
 - Пример простой программы на ассемблере
 - Разбор программы
-

Возможно, вас несколько утомили пространные экскурсы в историческую и архитектурную области. Многое вам, возможно, осталось непонятным, особенно, если это первая книга об архитектуре компьютера и его внутреннем языке, которую вы читаете. Смею вас уверить, что это никак не является поводом к тому, чтобы отложить чтение. Всем известно, что здание прочно стоит на хорошем фундаменте. В нашем случае фундаментом являются знания о компьютере. Естественно, что чем фундаментальнее и богаче они будут, тем более надежную и эффективную программу мы сможем построить. Именно это и побуждает нас столь подробно рассматривать вопросы архитектуры, хотя все можно было свести к обсуждению набора команд и основных приемов программирования, в итоге получив карточный домик со всеми вытекающими из этого последствиями.

Ободрившись этими замечаниями, посмотрим, что нас ждет дальше. Чтобы оживить повествование, мы приведем пример простой, но полноценной программы на ассемблере. В качестве преамбулы к постановке задачи обсудим одну проблему. На уроке 2 при обсуждении архитектуры мы перечислили большое количество регистров. Как правило, большинство из них задействовано при работе практически любой программы. Было бы интересно во время работы программы посмотреть их содержимое. Это нетрудно, если использовать специальную программу — отладчик. Но как сделать это динамически, не используя других программ? Или, к примеру, как решить обратную задачу — ввести с клавиатуры значения в регистр? Можно, конечно, написать соответствующую программу. Тот, кто работал на одном из современных языков высокого уровня, скажет: «Подумаешь, я вызову функцию, предназначенную для вывода содержимого регистра, и нет проблем». Действительно, проблем нет, если нас не интересует эффективность кода. Нельзя забывать, что между «железом» компьютера и любым языком высокого уровня стоит компилятор, который может генерировать, мягко говоря, не очень эффективный код. Для критичных по размеру системных программ, при написании которых порой учитывается каждый байт, практика использования исключительно языков высокого уровня выглядит сомнительной. На помощь приходит ассемблер. Если представление в машинном виде программы на языке высокого уровня — это черный ящик (в том смысле, что мы мало в чем можем повлиять на компиляцию), то, применяя язык ассемблера, мы можем учитывать тончайшие системные и архитектурные нюансы. Именно поэтому не следует при-
нижать значение низкоуровневого программирования по сравнению с мощными языками высокого уровня. Разработчики компиляторов по сей день обязательно оставляют возможность выхода на уровень ассемблера. Эта возможность может быть реализована в форме ассемблерных вставок в программу или подключения

внешних процедур на ассемблере. В любом случае это позволяет повысить качество получающегося кода.

Если вы убедились в полезности (естественно, в разумных пределах) использования ассемблера при разработке ваших программ, то продолжим разбираться с нашей проблемой, которая заключается в том, что нам требуется визуализировать содержимое некоторого регистра или ввести в него значение. При этом мы сразу столкнемся с другой частной проблемой — преобразования данных. Причина здесь в том, что компьютер понимает только те типы данных, которые поддерживаются его системой команд. Поэтому на практике часто возникает необходимость преобразования данных из одного представления в другое. На данном уроке мы только наметим контуры решения этой задачи и разберем частный случай.

Для начала нужно продумать алгоритм. Ввод информации с клавиатуры и вывод ее на экран осуществляются в символьном виде. Кодирование этой информации производится согласно *таблице ASCII*. В таблице ASCII каждый символ кодируется одним байтом. Если работа происходит с числами, то при попытке их обработать сразу возникает проблема: команд для арифметической обработки чисел в символьном виде нет. Что делать? Выход очевиден: нужно преобразовать символьную информацию к формату, поддерживаемому машинными командами. После такого преобразования нужно выполнить необходимые вычисления и преобразовать результат обратно к символьному виду. Затем следует отобразить информацию на мониторе.

На уроке 2 мы обсуждали логические типы данных, поддерживаемые на уровне машинных команд. Из приведенного перечня видно, что поддерживаются арифметические операции над двоичными числами, со знаком и без знака, и операции над десятичными числами. Десятичные числа, другое их название BCD-числа (Binary Code Decimal — двоично-десятичный код), наиболее удобны для программирования прикладных задач. Недостаток этих чисел в том, что для них требуется разработка специфических алгоритмов. Сейчас мы рассмотрим преобразование шестнадцатеричного числа из двух цифр, вводимого с клавиатуры (то есть в символьном виде), к двоичному виду. После выполнения этой операции полученное число можно использовать как operand в двоичных арифметических операциях.

Программа, представленная в листинге 3.1, является одним из вариантов решения этой задачи. В основу ее алгоритма положена особенность, связанная с ASCII-кодами символов соответствующих шестнадцатеричных цифр. Шестнадцатеричные цифры формируются из символов 0, 1, ..., 9, A, B, C, D, E, F, a, b, c, d, e, f, например: 12Af, 34ad.

В таблицах кодов ASCII Справочника можно найти значения кодов ASCII, соответствующие этим символам. На первый взгляд, непонятна популярность способа представления информации в виде шестнадцатеричных чисел. Мы уже знаем, что аппаратура компьютера построена на логических микросхемах и работает только с двоичной информацией. Если нам требуется проанализировать, например, состояние некоторой области памяти, то разобраться в нагромождении нулей

и единиц будет очень непросто. Для примера рассмотрим двоичную последовательность:

010100010101011110101101110101010101000101001010

Это представление не очень наглядно. Мы говорили, что оперативная память состоит из ячеек — байтов — по 8 битов. Приведенная выше цепочка битов при разбиении ее на байты будет выглядеть так:

01010001 01010111 10101101 11010101 01010001 01001010

С наглядностью стало лучше, но если область памяти будет больше, то разобраться будет все равно сложно. Давайте проведем еще одну операцию: каждый байт разобьем на две части по 4 бита — *тетрады*:

0101 0001 0101 0111 1010 1101 1101 0101 0101 0001 0100 1010

И вот тут проявляется замечательное свойство шестнадцатеричных чисел — каждой тетраде можно поставить в соответствие одну шестнадцатеричную цифру (табл. 3.1).

Таблица 3.1. Кодировка шестнадцатеричных цифр

| Символ шестнадцатеричной цифры | ASCII-код (двоичное представление) | Двоичная тетрада |
|--------------------------------|------------------------------------|------------------|
| 0 | 30h (0011 0000) | 0000 |
| 1 | 31h (0011 0001) | 0001 |
| 2 | 32h (0011 0010) | 0010 |
| 3 | 33h (0011 0011) | 0011 |
| 4 | 34h (0011 0100) | 0100 |
| 5 | 35h (0011 0101) | 0101 |
| 6 | 36h (0011 0110) | 0110 |
| 7 | 37h (0011 0111) | 0111 |
| 8 | 38h (0011 1000) | 1000 |
| 9 | 39h (0011 1001) | 1001 |
| A, a | 41h (0100 0001), 61h (0110 0001) | 1010 |
| B, b | 42h (0100 0010), 62h (0110 0010) | 1011 |
| C, c | 43h (0100 0011), 63h (0110 0011) | 1100 |
| D, d | 44h (0100 0100), 64h (0110 0100) | 1101 |
| E, e | 45h (0100 0101), 65h (0110 0101) | 1110 |
| F, f | 46h (0100 0110), 66h (0110 0110) | 1111 |

Если заменить в последней полученной нами строке тетрады на соответствующие шестнадцатеричные цифры, то получим последовательность 51 57 ad d5 51 8a:

0101 0001 0101 0111 1010 1101 1101 0101 0101 0001 0100 1010

5 1 5 7 a d d 5 5 1 8 a

Каждый байт наглядно представляется двумя шестнадцатеричными цифрами. Если привыкнуть к такой записи, то она оказывается очень удобной при работе с памятью, дисками и т. д.

Вернемся к нашей задаче. В табл. 3.1 приведены шестнадцатеричные цифры и их кодировка в коде ASCII.

Посмотрите внимательно и сравните эти представления. К примеру, рассмотрим шестнадцатеричный 0. Соответствующий ему код ASCII — 30h, который в двоичном представлении записывается как 0011 0000, а соответствующее двоичное число совпадает с двоичным представлением нуля 0000 0000.

Первый вывод: для шестнадцатеричных цифр 0...9 код ASCII отличается от соответствующего двоичного представления на 0011 0000, или 30h. Поэтому для преобразования кода ASCII в шестнадцатеричное число есть два пути:

1. Выполнить двоичное вычитание: (код ASCII)h — 30h.
2. Обнулить старшую тетраду байта с символом шестнадцатеричной цифры в коде ASCII.

Видно, что с шестнадцатеричными цифрами в диапазоне от 0 до 9 все просто. Что касается шестнадцатеричных цифр a, b, c, d, e, f, то здесь ситуация несколько сложнее. Алгоритм должен распознавать эти символы и производить дополнительные действия при их преобразовании в соответствующее двоичное число. Посмотрите внимательно на символы шестнадцатеричных цифр и соответствующие им двоичные представления (см. табл. 3.1). Видно, что для преобразования уже недостаточно простого вычитания или обнуления старшей тетрады. При анализе таблицы ASCII видно, что символы прописных букв шестнадцатеричных цифр отличаются от своего двоичного эквивалента на величину 37h. Соответствующие строчные буквы шестнадцатеричных цифр отличаются от своего двоичного эквивалента на 67h.

Отсюда следует *второй вывод*: алгоритм преобразования должен различать прописные и строчные буквенные символы шестнадцатеричных цифр и корректировать значение кода ASCII на величину 37h или 67h.

Вы, наверное, заметили, что после записи значения шестнадцатеричной цифры следует символ «h». Это сделано для того, чтобы транслятор мог отличить в программе одинаковые по форме записи десятичные и шестнадцатеричные числа. К примеру, числа 1578 и 1578h выглядят одинаково, но имеют разные значения. Более того, как вы думаете, какое значение в тексте исходной программы имеет лексема fe023? Это может быть и некоторый идентификатор, и, судя по набору символов, шестнадцатеричное число. Для того чтобы однозначно описать в тексте программы на ассемблере подобное шестнадцатеричное число, его дополняют ведущим нулем и в конце ставят «h». В нашем примере это будет выглядеть так: 0fe023h.

Разберем программу, представленную в листинге 3.1. Если у вас все еще остались неясные моменты с представлениями чисел в различных форматах, то не отчайвайтесь, так как на уроке 6 эти вопросы будут рассмотрены более систематизированно.

Листинг 3.1. Пример программы на ассемблере

```
<1> ;——Prg_3_1.asm—————  
<2> ;Программа преобразования двухзначного шестнадцатеричного числа  
<3> ;в символьном виде в двоичное представление.  
<4> ;Вход: исходное шестнадцатеричное число из двух цифр.  
<5> ;вводится с клавиатуры.  
<6> ;Выход: результат преобразования помещается в регистр d1.  
<7> ;  
<8> ;  
<9> data segment para public "data" ;сегмент данных  
<10> message db "Введите две шестнадцатеричные цифры,$"  
<11> data ends  
<12> stk segment stack  
<13>     db 256 dup (?) ;сегмент стека  
<14> stk ends  
<15> code segment para public "code" ;начало сегмента кода  
<16> main proc ;начало процедуры main  
<17> assume cs:code,ds:data,ss:stk  
<18> mov ax,data ;адрес сегмента данных в регистр ax  
<19> mov ds,ax ;ax в ds  
<20> mov ah,9  
<21> mov dx,offset message  
<22> int 21h  
<23> xor ax,ax ;очистить регистр ax  
<24> mov ah,1h ;1h в регистр ah  
<25> int 21h ;генерация прерывания с номером 21h  
<26> mov d1,a1 ;содержимое регистра al в регистр d1  
<27> sub d1,30h ;вычитание: (d1)=(d1)-30h  
<28> cmp d1,9h ;сравнить (d1) с 9h  
<29> jle M1 ;перейти на метку M1, если d1<9h или d1=9h  
<30> sub d1,7h ;вычитание: (d1)=(d1)-7h  
<31> M1: ;определение метки M1  
<32> mov c1,4h ;пересылка 4h в регистр c1  
<33> shl d1,c1 ;сдвиг содержимого d1 на 4 разряда влево  
<34> int 21h ;вызов прерывания с номером 21h  
<35> sub a1,30h ;вычитание: (d1)=(d1)-30h  
<36> cmp a1,9h ;сравнить (a1) с 9h    28  
<37> jle M2 ;перейти на метку M2, если a1<9h или a1=9h  
<38> sub a1,7h ;вычитание: (a1)=(a1)-7h  
<39> M2: ;определение метки M2  
<40> add d1,a1 ;сложение: (d1)=(d1)+(a1)  
<41> mov ax,4c00h ;пересылка 4c00h в регистр ax  
<42> int 21h ;вызов прерывания с номером 21h  
<43> main endp ;конец процедуры main  
<44> code ends ;конец сегмента кода  
<45> end main ;конец программы с точкой входа main
```

Подробно структура и правила оформления программ будут изложены на уроке 5. Сейчас мы рассмотрим эти правила лишь в части, касающейся нашего примера. На уроке 2 мы выяснили, что язык ассемблера является символическим представлением машинного языка. Поэтому естественным представляется то обстоятельство, что сам язык и программы на нем отражают архитектуру компьютера.

Посмотрите внимательно на программу. В ней есть три участка, заключенные между директивами `segment` и `ends` (строки 9 и 11, 12 и 14, 15 и 44). Микропроцессор аппаратно поддерживает шесть адресно-независимых областей памяти: сегмент кода, сегмент данных, сегмент стека и три дополнительных сегмента данных. Наша программа использует только первые три из них. Таким образом, уже сама структура программы отражает особенности организации памяти.

Строки 9–11 определяют сегмент данных. В строке 10 описана текстовая строка с сообщением «Введите две шестнадцатеричные цифры».

Строки 12–14 описывают сегмент стека, который является просто областью памяти длиной 256 байт, инициализированной символами ««?»». Отличие сегмента стека от сегментов других типов состоит в использовании и адресации памяти. В отличие от сегмента данных, — наличие которого необязательно, если программа не работает с данными, — сегмент стека желательно определять всегда.

Строки 15–44 содержат сегмент кода. В этом сегменте в строках 16–43 определена одна процедура `main`.

Строка 17 содержит директиву ассемблера, которая связывает сегментные регистры с именами сегментов.

Строки 18–19 выполняют инициализацию сегментного регистра `ds`.

Строки 20–22 выводят на экран сообщение `message`:

Введите две шестнадцатеричные цифры

Строка 23 готовит регистр `ax` к работе, обнуляя его. Содержимое `ax` после этой операции следующее:

`ax = 0000 0000 0000 0000`

Строки 24–25 обращаются к средствам операционной системы для ввода символа с клавиатуры. Введенный символ операционная система помещает в регистр `a1`. К примеру, в ответ на сообщение вы ввели с клавиатуры две шестнадцатеричные цифры:

5C

В результате после отработки команды в строке 25 будет введен один символ в коде ASCII — 5, и состояние регистра `ax` станет таким:

`ax = 0000 0001 0011 0101`

Строка 26 пересыпает содержимое `a1` в регистр `d1`. Это делается для того, чтобы освободить `a1` для ввода второй цифры. Содержимое регистра `dx` после этой пересылки следующее:

`dx = 0000 0000 0011 0101`

Строка 27 преобразует символьную 5 в ее двоичный эквивалент путем вычитания 30h, в результате чего в регистре d1 будет двоичное значение числа 5:

dx = 0000 0000 0000 0101

Строки 28–29 выясняют, нужно ли корректировать двоичное значение в d1. Если оно в диапазоне 0...9, то в d1 — правильный двоичный эквивалент введенного символа шестнадцатеричной цифры. Если значение в d1 больше 9, то введенная цифра является одним из символов A, B, C, D, E, F (строчные буквы для экономии места обрабатывать не будем). В первом случае строка 29 передаст управление на метку M1. При обработке цифры 5 это условие как раз выполняется, поэтому происходит переход на метку M1 (строка 31).

Каждая шестнадцатеричная цифра занимает одну тетраду. У нас — две таких цифры, поэтому нужно их разместить так, чтобы старшинство разрядов сохранилось. Строки 32–33 сдвигают значение в d1 на 4 разряда влево, тем самым освобождая место в младшей тетраде под младшую шестнадцатеричную цифру.

Строка 34 вводит вторую шестнадцатеричную цифру С (ее ASCII-код 63h) в регистр a1:

ax = 0000 0001 0100 0011

Строки 35–37 выясняют, попадает ли двоичный эквивалент второго символа шестнадцатеричной цифры в диапазон 0...9. Наша вторая цифра не попадает в диапазон, поэтому для получения правильного двоичного эквивалента нужно произвести дополнительную корректировку. Это делает строка 38. Состояние a1 после выполнения строки 35 следующее:

ax = 0000 0001 0001 0011

В a1 записано 13h, а нужно, чтобы было 0Ch (помните о правилах записи шестнадцатеричных чисел!). Так как 0Ch не попадает в диапазон 0...9, то происходит переход на строку 38. В результате работы команды вычитания в регистре a1 получается правильное значение (a1) = 0Ch:

ax = 0000 0001 0000 1100

И наконец, строка 40 производит сложение сдвинутого значения в d1 с числом в a1:

dx = 0000 0000 0101 0000

+

ax = 0000 0001 0000 1100

=

dx = 0000 0000 0101 1100

Таким образом, в регистре d1 мы получили двоичный эквивалент двух введенных символов, изображающих двузначное шестнадцатеричное число:

(d1) = 05Ch

Строки 41–42 предназначены для завершения программы и возврата управления операционной системе.

Мы не случайно столь детально рассмотрели этот пример. Он отражает практически все специфические особенности программирования на ассемблере. Остался неясным вопрос о том, как организовать выполнение данной программы. Этим мы и займемся на следующем уроке.

Подведем некоторые итоги:

- Структура программы на ассемблере отражает особенности архитектуры процессора. Для микропроцессоров Intel типичная программа состоит из трех сегментов: кода, стека и данных. Хотя, как вы понимаете, это не обязательно. Если ваша программа не использует стек и для ее работы не требуется определения данных, то она может состоять из одного сегмента кода.
- Программа на ассемблере работает на уровне аппаратных средств, входящих в программную модель микропроцессора, с которой мы познакомились на прошлом уроке.
- При разработке алгоритма программы и его реализации на ассемблере программист сам должен беспокоиться о размещении данных в памяти, об эффективном использовании ограниченного количества регистров, об организации связи с операционной системой и другими программами.
- Из-за того что компьютер на самом нижнем уровне работает только с двоичной информацией, программисту необходимо ее понимать. Для этого ему нужно научиться работать с шестнадцатеричным представлением чисел. В прикладных программах часто используют десятичные числа. Для обмена с устройствами ввода/вывода используют кодировку символов по таблице ASCII. Подробно о системах счисления мы поговорим на уроке 6.

4

УРОК

Жизненный цикл программы на ассемблере

- Жизненный цикл программы на ассемблере
- Разработка программ на ассемблере с использованием пакета TASM
- Назначение и структура выходных файлов, формируемых транслятором
- Трансляция и компоновка программы урока 3
- Отладка программы из урока 3
- Менеджер проекта — утилиты MAKE

На этом уроке мы познакомимся со специальными программными средствами, предназначенными для преобразования исходных текстов на ассемблере к виду, приемлемому для выполнения на компьютере, и научимся использовать их.

Но прежде чем обсуждать сами инструментальные средства разработки программ, необходимо уделить внимание общим методологическим принципам разработки программного обеспечения. Если вы — начинающий программист, то у вас наверняка очень большой интерес к практической работе и, возможно, разработку программы вы производите на чисто интуитивном уровне. До определенного момента здесь нет ничего страшного; это даже естественно. Но совсем не задумываться над тем, как правильно организовать разработку программы (не обязательно на ассемблере), нельзя, так как хаотичность и ставка только на интуицию в конечном итоге станут стилем программирования. А это может привести к тому, что рано или поздно за вами закрепится слава программиста, у которого программы работают «почти всегда» со всеми вытекающими отсюда последствиями для вашей карьеры. Поэтому, как мне кажется, нужно помнить одно золотое правило: *надежность программы достигается, в первую очередь, благодаря ее правильному проектированию, а не бесконечному тестированию.*

Это правило означает, что если программа правильно разработана в отношении как структур данных, так и структур управления, то это в определенной степени гарантирует правильность ее функционирования. При применении такого стиля программирования ошибки являются легко локализуемыми и устранимыми.

О том, как правильно организовать разработку программ (независимо от языка), написана не одна сотня книг. Большинство авторов предлагает следующий процесс разработки программы (мы адаптируем его, где это необходимо, к особенностям ассемблера):

1. Этап постановки и формулировки задачи:

- изучение предметной области и сбор материала в проблемно-ориентированном контексте;
- определение назначения программы, выработка требований к ней и представление требований, если возможно, в формализованном виде;
- формулирование требований к представлению исходных данных и выходных результатов;
- определение структур входных и выходных данных;
- формирование ограничений и допущений на исходные и выходные данные.

2. Этап проектирования:

- формирование «ассемблерной» модели задачи;
- выбор метода реализации задачи;
- разработка алгоритма реализации задачи;
- разработка структуры программы в соответствии с выбранной моделью памяти.

3. Этап кодирования:

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

4. Этап отладки и тестирования:

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

5. Этап эксплуатации и сопровождения:

- настройка программы на конкретные условия использования;
- обучение пользователей работе с программой;
- организация сбора сведений о сбоях в работе программы, ошибках в выходных данных, пожеланиях по улучшению интерфейса и удобства работы с программой;
- модификация программы с целью устранения выявленных ошибок и, при необходимости, изменения ее функциональных возможностей.

К порядку применения и полноте выполнения перечисленных этапов нужно подходить разумно. Многое определяется особенностями конкретной задачи, ее назначением, объемом кода и обрабатываемых данных, другими характеристиками задачи. Некоторые из этих этапов могут либо выполняться одновременно с другими этапами, либо вовсе отсутствовать. Главное, чтобы вы, приступая к созданию нового программного продукта, помнили о необходимости его концептуальной целостности и недопустимости анархии в процессе разработки.

На уроке 3 мы обсуждали пример программы на ассемблере. Если посмотреть на описанный выше процесс разработки программы, то можно увидеть, что обсуждение на уроке 3 велось нами в полном согласии с этим процессом. Мы подробно обсудили проблему, структуры данных, структуру программного модуля и т. д. Наше обсуждение закончилось на этапе кодирования программы. Далее, по логике, нужно было ввести программу в компьютер, перевести в машинное представление и выполнить. Как это сделать? Дальнейшее обсуждение будет посвящено именно этому вопросу.

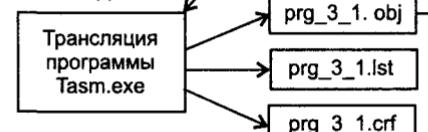
У большинства существующих реализаций ассемблера нет интегрированной среды, подобной интегрированным средам Turbo Pascal, Turbo C или Visual C++. Поэтому для выполнения всех функций по вводу кода программы, ее трансляции, редактированию и отладке необходимо использовать отдельные служебные программы. Большая часть их входит в состав специализированных *пакетов* ассемблера.

На рис. 4.1 приведена общая схема процесса разработки программы на ассемблере на примере программы урока 3 (см. листинг 3.1). На схеме выделено четыре шага этого процесса. На первом шаге, когда вводится код программы, можно использовать любой текстовый редактор. Основным требованием к нему является то, чтобы он не вставлял посторонних символов (специальных символов форматирования). Файл должен иметь расширение .asm.

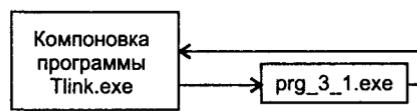
1. ВВОД ИСХОДНОГО ТЕКСТА ПРОГРАММЫ



2. СОЗДАНИЕ ОБЪЕКТНОГО МОДУЛЯ



3. СОЗДАНИЕ ЗАГРУЗОЧНОГО МОДУЛЯ



4. ОТЛАДКА ПРОГРАММЫ

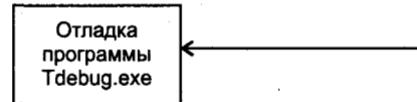


Рис. 4.1. Процесс разработки программы на ассемблере

Программы, реализующие остальные шаги схемы, входят в состав программного пакета ассемблера. Большую популярность на рынке ассемблеров для микропроцессоров фирмы Intel имеют два пакета:

- «Макроассемблер» MASM фирмы Microsoft;
- Turbo Assembler TASMS фирмы Borland.

У этих пакетов много общего. Пакет макроассемблера фирмы Microsoft (MASM) получил свое название потому, что он позволяет программисту задавать *макропределения* (или *макросы*), представляющие собой именованные группы команд.

Они обладают тем свойством, что их можно вставлять в программу в любом месте, указав только имя группы в месте вставки. Пакет Turbo Assembler (TASM) интересен тем, что имеет два режима работы. Один из этих режимов, называемый MASM, поддерживает все основные возможности макроассемблера MASM. Другой режим, называемый IDEAL, предоставляет более удобный синтаксис написания программ, более эффективное использование памяти при трансляции программы и другие новшества, приближающие компилятор ассемблера к компиляторам языков высокого уровня.

В эти пакеты входят трансляторы, компоновщики, отладчики и другие утилиты для повышения эффективности процесса разработки программ на ассемблере. Перечислим и дадим краткую характеристику следующим наиболее интересным средствам пакета TASM:

- 16- и 32-разрядные трансляторы `tasm.exe` и `tasm32.exe`;
- 16- и 32-разрядные компоновщики (редакторы связей) `tlink.exe` и `tlink32.exe`;
- Turbo Debugger (TD) — отладчик, работающий на уровне исходного текста. Имеет 16- и 32-разрядные версии `td.exe` и `td32.exe`. Существует отладчик `tdw.exe`, позволяющий производить отладку Windows-приложений. В комплексе с TD поставляется ряд дополнительных утилит:
 - `tdstrip.exe` (`tdstrip32.exe`) — утилита для удаления отладочной информации из исполняемого файла без его перекомпиляции, что приводит к существенному уменьшению его размера. Но лучше все же заново перекомпилировать исходный файл, и его размер станет еще меньше. Это говорит о том, что утилита `tdstrip.exe` удаляет не всю отладочную информацию. Утилита `tdstrip32.exe` представляет 32-разрядный вариант утилиты `tdstrip.exe`, работающей с исполняемыми файлами, созданными `tasm32.exe` и `tlink32.exe`;
 - `tdwini.exe` — утилита для установки dll-библиотеки, обеспечивающей корректную работу отладчика `tdw.exe` с видеосредствами компьютера;
- `tdump.exe` — утилита, позволяющая получить информацию о структуре и содержании исполняемого файла. Например, для получения дампа программы `myprog.exe` введите в командной строке:
`tdump.exe myprog.exe>r.txt`
- В результате работы утилиты будет создан файл `r.txt`, содержащий достаточно интересную для последующего анализа информацию об исполняемом файле `myprog.exe`;
- `implib.exe` — утилита создания библиотек импорта. Данные библиотеки нужны компоновщику для разрешения ссылок из программы на функции, находящиеся в dll-библиотеках. Более подробно эта тема обсуждается на уроке 18;
- `impdef.exe` — утилита, формирующая файл определений `.def` с экспортной секцией. На вход этой утилиты подается файл с расширением `.dll`. В результате работы утилиты формируется файл с расширением `.def`, содержащий секцию со списком функций, экспортируемых dll-библиотекой;
- `tlib.exe` — утилита-библиотекарь. Данная утилита позволяет вести библиотеку, которая предназначена для хранения наиболее часто используемых объектных

модулей. Библиотечный файл может подаваться на вход редактора связей для компоновки с другими объектными модулями;

- **h2ash.exe** и **h2ash32.exe** — утилиты преобразования включаемых файлов C/C++ (с расширением .h) во включаемые файлы TASM (с расширением .ash или .inc). Утилита удобна при разработке Windows-приложений;
- **brc32.exe** и **brcc32.exe** — компиляторы ресурсов. Подробнее материал об использовании компилятора ресурсов изложен на уроке 18.

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

Мы воспользуемся тем, что транслятор TASM, работая в режиме MASM, поддерживает почти все возможности транслятора MASM. Для работы с данной книгой необходимо иметь пакет ассемблера фирмы Borland — TASM 5.0 или выше. Обратившись к этому пакету, мы «убьем сразу двух зайцев» — изучим основы и TASM, и MASM. В будущем это позволит вам при необходимости использовать любой из этих пакетов.

Трансляция программы

Итак, исходный текст программы на ассемблере подготовлен и записан на диск. Следующий шаг — *трансляция* программы. На этом шаге формируется *объектный модуль*, который включает в себя представление исходной программы в машинных кодах и некоторую другую информацию, необходимую для отладки и компоновки его с другими модулями. Для получения объектного модуля исходный файл необходимо подвергнуть трансляции при помощи программы **tasm.exe** из пакета TASM. Формат командной строки для запуска **tasm.exe** следующий:

**TASM [опции] имя_исходного_файла [.имя_объектного_файла]
[.имя_файла_листинга] [.имя_файла_перекрестных_ссылок]**

На первый взгляд, все очень сложно. Не пугайтесь — если вы вдруг забыли формат командной строки и возможные значения параметров, то получить быструю справку на экране монитора можно, просто запустив **tasm.exe** без задания каких-либо аргументов. Обратите внимание, что большинство параметров заключено в квадратные скобки. Это общепринятое соглашение по обозначению параметров, которые могут отсутствовать. Таким образом, обязательным аргументом командной строки является лишь **имя_исходного_файла**. Этот файл должен находиться на диске и обязательно иметь расширение .asm. За именем исходного файла через запятую могут следовать необязательные аргументы, обозначающие имена объектного файла, файла листинга и файла перекрестных ссылок. Если не задать их, то соответствующие файлы попросту не будут созданы. Если же их нужно создать, то необходимо учитывать следующее:

- Если имена объектного файла, файла листинга и файла перекрестных ссылок должны совпадать с именем исходного файла (наиболее типичный случай), то нужно просто поставить запятые вместо имен этих файлов:

tasm.exe prg_3_1 . . .

В результате будут созданы файлы, как показано на рис. 4.1 для шага 2.

- Если имена объектного файла, файла листинга и файла перекрестных ссылок не должны совпадать с именем исходного файла, то нужно в соответствующем порядке в командной строке указать имена соответствующих файлов, к примеру:

`tasm.exe prg_3_1 . .prg_list .`

В результате на диске будут созданы файлы

`prg_3_1.obj
prg_list.lst
prg_3_1.crf`

- Если требуется выборочное создание файлов, то вместо ненужных файлов необходимо подставить параметр `nul`. Например:

`tasm.exe prg_3_1 . .nul.`

В результате на диске будут созданы файлы

`prg_3_1.obj
prg_3_1.crf`

Необязательный аргумент опции позволяет задавать режим работы транслятора TASM. Этих опций достаточно много, и все они описаны в Справочнике. Некоторые из опций понадобятся нам в ближайшее время, а большинство из них, скорее всего, никогда не будут вами востребованы.

У вас наверняка уже возникло множество вопросов. Большая часть их снимется по мере прочтения книги, работы с материалом и накопления опыта.

Давайте немного поэкспериментируем с программой `tasm.exe`. Попутно мы выясним еще несколько важных моментов. Прежде всего, проведем некоторые организационные мероприятия. После инсталляции пакета TASM в каталоге `\TASM\BIN`, где находится файл `tasm.exe`, вы увидите большое количество файлов. Можно запустить программу `tasm.exe` прямо отсюда, но тогда созданные ею файлы объектного кода, листинга и перекрестных ссылок тоже окажутся в этом каталоге. Если вы пишете одну программу, то неудобство не столь заметно, но при работе с несколькими программами очень скоро этот каталог станет похож на свалку. Чтобы избежать подобной ситуации, рекомендуется выполнить следующие действия:

- Создать в каталоге `\TASM` подкаталоги `\TASM\WORK` и `\TASM\PROGRAM`. Каталог `PROGRAM` будем использовать для хранения отлаженных кодов программ и их исполняемых модулей (файлы с расширением `.exe`). Каталог `WORK` будем использовать как рабочий; в нем будут находиться необходимые для получения исполняемого модуля файлы из пакета транслятора TASM и файл исходного модуля, с которым мы в данный момент работаем. После того как ошибки в исходном модуле устраниены, он вместе со своим исполняемым модулем переписывается в каталог `PROGRAM`. Из каталога `WORK` удаляются все ненужные файлы — и он готов для работы со следующим исходным модулем на ассемблере. Таким образом, в каталоге `WORK` всегда находится рабочая версия программы, а в каталоге `PROGRAM` — отложенная версия.

- Поместить в каталог WORK файлы **tasm.exe**, **tlink.exe** и **rtm.exe**. Если вы что-то забудете туда поместить, программы **tasm.exe** и **tlink.exe** выдадут вам сообщение об этом.
- Поместить файл **prg_3_1.asm** в каталог WORK.

После всех этих действий можно начинать работу. Переходим в каталог WORK и запустим на трансляцию программу **prg_3_1.asm** командной строкой вида:

tasm.exe /zi prg_3_1 . . .

В результате на экране вы получите последовательность строк. Самая первая из них будет информировать вас о номере версии пакета TASM, который использовался для трансляции данной программы. Далее идет строка, содержащая имя транслируемого файла. Если ваша программа содержит ошибки, то транслятор выдаст на экран строки сообщений, начинающиеся словами «**Error**» и «**Warning**». Программа урока 3 (см. листинг 3.1) синтаксически правильная, но в учебных целях вы можете внести туда какую-нибудь бессмыслицу и посмотреть, что получится. Наличие строки с «**Error**» говорит о том, что у вас в программе есть недопустимые, с точки зрения синтаксиса, комбинации символов. Логика работы программы для транслятора не имеет никакого значения. Вы можете написать абсолютную чушь, но если она будет синтаксически правильна, транслятор поспешит вас обрадовать, сообщив, что все хорошо. Наличие строки «**Warning**» означает, что конструкция синтаксически правильна, но не соответствует некоторым соглашениям языка, и это может послужить источником последующих ошибок. Для устранения ошибок нужно определить место их возникновения и проанализировать ситуацию. Место ошибки легко определяется по значению в скобках в сообщении об ошибке. Это значение является номером ошибочной строки. Запомнив его, вы переходите в файл с исходной программой и по номеру строки находите место ошибки.

Этот способ локализации ошибок имеет недостатки. Во-первых, он не нагляден. Во-вторых, не всегда номера строк в сообщении соответствуют действительным номерам ошибочных строк в исходном файле. Такая ситуация будет наблюдаться, например, в том случае, если вы используете макрокоманды. При их использовании транслятор вставляет в файл дополнительные строки в соответствии с описанием применяемой макрокоманды, в результате чего получается отличие в нумерации. Исходя из этих соображений, для локализации ошибок лучше использовать информацию из специального, создаваемого транслятором *файла листинга*. Этот файл имеет расширение **.lst**; его имя определяется в соответствии с теми соглашениями, которые мы разобрали выше. Ниже приведен полный формат листинга для программы, содержащей некоторые ошибки. Листинг – очень важный документ, и ему нужно уделить должное внимание.

Листинг 4.1. Пример листинга ассемблера

Turbo Assembler Version 4.1 02/03/98 21:23:43 Page 1

Prg_3_1.asm

- 1 :—Prg_3_1.asm————
- 2 :Программа преобразования двузначного шестнадцатеричного числа
- 3 :в символьном виде в двоичное представление.
- 4 :Вход: исходное шестнадцатеричное число из двух цифр.

5 ;вводится с клавиатуры.
6 ;Выход: результат помещается
7 ;в регистр a1.
8 :
9 0000 data segment para public "data" ;сегмент данных
10 0000 82 A2 A5 A4 A8 E2 A5+ message db "Введите две шестнадцатеричные
цифры,\$"
11 20 A4 A2 A5 20 E8 A5+
12 E1 E2 AD A0 A4 E6 A0+
13 E2 A5 E0 A8 E7 AD EB+
14 A5 20 E6 A8 E4 E0 EB+
15 2C 24
16 0025 data ends
17 0000 stk segment stack
18 0000 0100*(3F) db 256 dup (?) ;сегмент стека
19 0100 stk ends
20 0000 code segment para public "code" ;начало сегмента кода
21 0000 main proc ;начало процедуры main
22 assume cs:code,ds:data,ss:stk
23 0000 B8 0000s mov ax,data ;адрес сегмента данных в регистр ax
24 0003 8E D8 mov ds,ax ;ax в ds
25 0005 B4 09 mov ah,9
26 0007 BA 0000 mov dx,offset messag
Error Prg_3_1.asm(21) Undefined symbol: MESSAG
27 000A CD 21 int 21h
28 000C 33 C0 xor ax,ax ;очистить регистр ax
29 000E B4 01 mov ah,1h ;1h в регистр ah
30 0010 CD 21 int 21h ;генерация прерывания с номером 21h
31 0012 8A D0 mov d1,a1 ;содержимое регистра a1 в регистр d1
32 0014 80 EA 30 sub d1,30h ;вычитание: (d1)=(d1)-30h
33 0017 80 FA 09 cmp d1,9h ;сравнить (d1) с 9h
34 001A 7E E4 jle MM ;перейти на метку M1, если d1<9h или d1=9h
Error Prg_3_1.asm(29) Undefined symbol: MM
35 001C 80 EA 00 sub d1,777h ;вычитание: (d1)=(d1)-7h
Error Prg_3_1.asm(30) Constant too large
36 001F M1: ;определение метки M1
37 001F B1 04 mov cl,4h ;пересылка 4h в регистр cl
38 0021 D2 E2 shl d1,cl ;сдвиг содержимого d1 на 4 разряда влево
39 0023 CD 21 int 21h ;вызов прерывания с номером 21h
40 0025 2C 30 sub a1,30h ;вычитание: (d1)=(d1)-30h
41 0027 3C 09 cmp a1,9h ;сравнить (a1) с 9h 28
42 0029 7E 02 jle M2 ;перейти на метку M2, если a1<9h или a1=9h
43 002B 2C 07 sub a1,7h ;вычитание: (a1)=(a1)-7h
44 002D M2: ;определение метки M2

```

45 002D 02 D0 add d1,a1 ;сложение: (d1)=(d1)+(a1)
46 002F B8 4C00 mov ax,4c00h ;пересылка 4c00h в регистр ах
47 0032 CD 21 int 21h ;вызов прерывания с номером 21h
48 0034 main endp      ;конец процедуры main
49 0034 code ends      ;конец сегмента кода
50 end main             ;конец программы с точкой входа main

```

Turbo Assembler Version 4.1 02/03/98 21:23:43 Page 2

Symbol Table

| Symbol Name | Type | Value | Cref (defined at #) |
|-------------|------|-------|---------------------|
|-------------|------|-------|---------------------|

```

??DATE Text "02/03/98"
??FILENAME Text "Prg_3_1"
??TIME Text "21:23:43"
??VERSION Number 040A
@CPU Text 0101H
@CURSEG Text CODE #9 #17 #20
@FILENAME Text PRG_3_1
@WORDSIZE Text 2 #9 #17 #20
M1 Near CODE:001F #36
M2 Near CODE:002D 42 #44
MAIN Near CODE:0000 #21 50
MESSAGE Byte DATA:0000 #10

```

| Groups & Segments | Bit | Size | Align | Combine | Class | Cref (defined at #) |
|-------------------|-----|------|-------|---------|-------|---------------------|
|-------------------|-----|------|-------|---------|-------|---------------------|

```

CODE    16 0034 Para Public CODE #20 22
DATA    16 0025 Para Public DATA #9 22 23
STK16 0100 Para Stack 17 22

```

Turbo Assembler Version 4.1 02/03/98 21:23:43 Page 3

Error Summary

```

**Error** Prg_3_1.asm(21) Undefined symbol: MESSAGE
**Error** Prg_3_1.asm(29) Undefined symbol: MM
**Error** Prg_3_1.asm(30) Constant too large

```

Файл листинга содержит, в частности, код ассемблера исходной программы. Но в листинге приводится расширенная информация об этом коде. Для каждой команды ассемблера указываются ее машинный (объектный) код и смещение в кодовом сегменте. Кроме того, в конце листинга TASM формирует таблицы, которые содержат информацию о метках и сегментах, используемых в программе. Если есть ошибки или сомнительные участки кода, то TASM включает в конец листинга сообщения о них. Если сравнить их с сообщениями, выводимыми на экран, то видно, что они совпадают. Кроме того, что очень удобно, эти же сообщения включаются в текст листинга непосредственно после ошибочной строки.

Строки в файле листинга имеют следующий формат:

<глубина_вложенности> <номер_строки> <смещение> <машинный_код> <исходный_код>

Здесь:

- <глубина_вложенности> — уровень вложенности включаемых файлов или макроКоманд в файле;
- <номер_строки> — номер строки в файле листинга. Эти номера используются для локализации ошибок и формирования *таблицы перекрестных ссылок*. Помните, что эти номера могут не соответствовать номерам строк в исходном файле. В добавление к вышесказанному нужно отметить, что ассемблер имеет директиву **INCLUDE**, которая позволяет включить в данный файл строки другого файла. Нумерация при этом, как и в случае макроКоманд, будет последовательная для строк обоих файлов. Факт вложенности кода одного файла в другой фиксируется увеличением значения <глубина_вложенности> на единицу. Это замечание касается и использования макроКоманд;
- <смещение> — смещение в байтах текущей команды относительно начала сегмента кода. Это смещение называют также *счетчиком адреса*. Величину смещения вычисляет транслятор для адресации в сегменте кода;
- <машинный_код> — машинное представление команды ассемблера, представленной далее в этой строке полем <исходный_код>;
- <исходный_код> — строка кода из исходного файла.

Дальнейшие ваши действия зависят от характера ошибки. По мере накопления опыта ошибки будут происходить, скорее всего, в результате простой описки. Пока же ваши действия будут заключаться в выяснении того, насколько правильно написана та или иная синтаксическая конструкция. Исправив несколько первых ошибок, перетранслируйте программу и приступайте к устранению следующих ошибок. Возможно, что этого делать не придется, так как после исправления одной ошибки могут исчезнуть и последующие (так называемые *наведенные ошибки*).

О нормальном окончании процесса трансляции можно судить по сообщению TASM, в котором отсутствуют строки с сообщениями об ошибках и предупреждениях.

Изучая внимательно файл листинга, вы, наверное, заметили, что не все строки исходной программы имеют соответствующий <машинный_код> (строки 9, 16, 17, 19...22, 50). Это обстоятельство обусловлено тем, что исходный файл на ассемблере в общем случае может содержать конструкции следующих типов:

- *команды ассемблера* — конструкции, которым соответствуют машинные команды;
- *директивы ассемблера* — конструкции, которые не генерируют машинных команд, а являются указаниями транслятору на выполнение некоторых действий или служат для задания режима его работы;
- *макроКоманды* — конструкции, которые, будучи представлены одной строкой в исходном файле программы, после обработки транслятором генерируют в объектном модуле последовательность команд, директив или макроКоманд ассемблера.

Формат листинга и его полнота не являются жестко регламентированными. Их можно изменить, задавая в исходном файле программы *директивы управления листингом*. Они подробно описаны в Справочнике.

Компоновка программы

После того как мы устранили ошибки и получили объектный модуль, можно приступить к следующему шагу — созданию исполняемого (загрузочного) модуля, или, как еще называют этот процесс, к компоновке программы. Главная цель этого шага — преобразовать код и данные в объектных файлах в их *перемещаемое выполняемое отображение*. Чтобы понять, в чем здесь суть, нужно разобраться, зачем вообще разделяют процесс создания исполняемого модуля на два шага — трансляцию и компоновку. Это сделано намеренно для того, чтобы можно было объединять вместе несколько модулей (написанных на одном или нескольких языках). Формат объектного файла позволяет, при определенных условиях, объединить несколько отдельно оттранслированных исходных модулей в один модуль. При этом в функции компоновщика входит разрешение внешних ссылок (ссылок на процедуры и переменные) в этих модулях. Результатом работы компоновщика является создание загрузочного файла с расширением .exe. После этого операционная система может загрузить такой файл в память и выполнить его.

Полный формат командной строки для запуска компоновщика достаточно сложен. В книге мы в основном будем использовать простой ее формат, но в последних уроках нам понадобятся значения практически всех параметров. Поэтому рассмотрим полный формат командной строки для запуска компоновщика:

```
TLINK [опции] список_объектных_файлов [.имя_загрузочного_модуля]  
[.имя_файла_карты] [.имя_файла_библиотеки] [.имя_файла_определений]  
[.имя_ресурсного_файла]
```

Здесь:

- **опции** — необязательные параметры, управляющие работой компоновщика. Список наиболее часто используемых опций приведен в Справочнике. Каждой опции должен предшествовать один из следующих символов: «-» или «/»;
- **список_объектных_файлов** — обязательный параметр, содержащий список компонуемых файлов с расширением .obj. Файлы должны быть разделены пробелами или знаком «+», например:

```
tlink /v prog + mdf + fdr
```

При необходимости имена файлов снабжают указанием пути к ним;

- **имя_загрузочного_модуля** — необязательный параметр, обозначающий имя формируемого загрузочного модуля. Если оно не указано, то имя загрузочного модуля будет совпадать с первым из списка имен объектных файлов;
- **имя_файла_карты** — необязательный параметр, наличие которого обязывает компоновщик создать специальный файл с картой загрузки. В ней перечисляются имена, адреса загрузки и размеры всех сегментов, входящих в программу;
- **имя_файла_библиотеки** — необязательный параметр, который представляет собой путь к файлу библиотеки (.lib). Этот файл создается и обслуживается специ-

альной утилитой `tlib.exe` пакета TASM. Утилита позволяет объединить часто используемые подпрограммы в виде объектных модулей в один файл. В дальнейшем вы можете указывать в командной строке `tlink.exe` имена нужных для компоновки объектных модулей и файл библиотеки, в котором следует искать эти подпрограммы. Если вы компонуете Windows-приложение, то на месте параметра `имя_файла_библиотеки` указывается имя библиотеки импорта (подробнее смотрите материал урока 18);

- `имя_файла_определений` — необязательный параметр, который представляет собой путь к файлу определений (`.def`). Этот файл используется при компоновке Windows-приложений (см. урок 18);
- `имя_ресурсного_файла` — необязательный параметр, который представляет собой путь к файлу с ресурсами Windows-приложения (`.res`). Этот файл используется при компоновке Windows-приложений (см. урок 18).

Рассмотренный нами формат командной строки используется и для 32-разрядного варианта компоновщика `tlink32.exe`.

Существует возможность задания параметров командной строки компоновщика в текстовом файле. Пусть создан файл с именем `tlink.cfg` (`tlink32.cfg`). Тогда при вызове компоновщика `tlink.exe` с параметром `tlink.cfg` (`tlink32.exe tlink32.cfg`), ему будет передано содержимое файла, например:

`/v`

`/Twe`

В результате применения такого файла будет создаваться Windows-приложение, загрузочный модуль которого содержит отладочную информацию. Полный перечень возможных опций `tlink.exe` (`tlink32.exe`) приведен в Справочнике. Обратите внимание, что при задании имен опций имеет значение регистр введенных символов.

Так же как и для синтаксиса `tasm.exe`, совсем не обязательно запоминать подробно синтаксис команды `tlink.exe`. Для того чтобы получить список опций программы `tlink.exe`, достаточно просто запустить ее без указания параметров.

Для выполнения нашего примера запустим программу `tlink.exe` командной строкой вида:

`tlink.exe /v prg_3_1.obj`

В результате вы получите исполняемый модуль с расширением `.exe` — `prg_3_1.exe`.

Получив исполняемый модуль, не спешите радоваться. К сожалению, устранение синтаксических ошибок еще не гарантирует того, что программа будет хотя бы запускаться, не говоря уже о правильности работы. Поэтому обязательным этапом процесса разработки является *отладка*.

На этапе отладки, используя описание алгоритма, выполняется контроль правильности функционирования как отдельных участков кода, так и всей программы в целом. Но даже успешное окончание отладки еще не является гарантией того, что программа будет работать правильно со всеми возможными исходными данными. Поэтому нужно обязательно провести *тестирование* программы, то есть проверить ее работу на «пограничных» и заведомо некорректных исходных данных.

Для этого составляются тесты. Вполне возможно, что результаты тестирования вас не удовлетворят. В этом случае придется вносить поправки в код программы, то есть возвращаться к первому шагу процесса разработки (см. рис. 4.1).

Специфика программ на ассемблере состоит в том, что они интенсивно работают с аппаратными ресурсами компьютера. Это обстоятельство заставляет программиста постоянно отслеживать содержимое определенных регистров и областей памяти. Естественно, что человеку трудно следить за этой информацией с большой степенью детализации. Поэтому для локализации логических ошибок в программах используют специальный тип программного обеспечения — **программные отладчики**.

Отладчики бывают двух типов:

- **интегрированные** — отладчик реализован в виде интегрированной среды типа среды для языков Turbo Pascal, Quick C и т. д.;
- **автономные** — отладчик представляет собой отдельную программу.

Из-за того что ассемблер не имеет своей интегрированной среды, для отладки написанных на нем программ используют автономные отладчики. К настоящему времени разработано большое количество таких отладчиков. В общем случае с помощью автономного отладчика можно исследовать работу любой программы, для которой создан исполняемый модуль, независимо от того, на каком языке был написан его исходный текст.

В этой книге будет рассмотрен отладчик Turbo Debugger (TD). Принципиально то, что основная информация о нем в той или иной степени относится и к другим отладчикам. Рассмотрим основные моменты работы с отладчиком TD.

Отладка программы

Отладчик Turbo Debugger (TD), разработанный фирмой Borland International, представляет собой оконную среду отладки программ на уровне исходного текста на языках Pascal, C, ассемблер. Он позволяет решить две главные задачи:

- определить место логической ошибки;
- определить причину логической ошибки.

Перечислим некоторые возможности TD:

- выполнение *трассировки* программы в прямом направлении, то есть последовательное исполнение программы, при котором за один шаг выполняется одна машинная инструкция;
- выполнение *трассировки* программы в обратном направлении, то есть выполнение программы по одной команде, но в обратном направлении;
- просмотр и изменение состояния аппаратных ресурсов микропроцессора во время покомандного выполнения программы.

Это позволяет определить место и источник ошибки в программе. Нужно сразу оговориться, что TD не позволяет вносить исправления в исходный текст программы. После определения причины ошибочной ситуации можно при необходимости, не завершая работу отладчика, внести исправления прямо в машинный код и запустить программу на выполнение. После завершения работы отладчика эти изменения не будут сохранены, и нужно внести их повторно, но уже в исходный текст, и повторно создать загрузочный модуль.

Как правильно организовать процесс получения исполняемого модуля, чтобы можно было выполнять его отладку на уровне исходного текста, мы уже рассмотрели выше. Вспомним ключевые моменты этого процесса:

- В исходной программе должна быть обязательно определена метка для первой команды, с которой начнется выполнение программы. Такая метка может быть собственно меткой или, как мы видели на примере программы из листинга 3.1, именем процедуры. Имя этой метки обязательно нужно указать в конце программы в качестве операнда директивы END:

```
END имя_метки
```

В нашем случае эта метка является именем процедуры `main`.

- Исходный модуль должен быть оттранслирован с опцией /zi:
`tasm /zi имя_исходного_модуля . . .`

Применение опции /zi разрешает транслятору сохранить связь символьических имен в программе и их смещений в сегменте кода, что позволит отладчику производить отладку, используя оригинальные имена.

- Редактирование модуля должно быть осуществлено с опцией /v:
`tlink /v имя_объектного_модуля`

Опция /v указывает на необходимость сохранения отладочной информации в исполняемом файле.

- Запуск отладчика удобнее производить из командной строки с указанием исполняемого модуля программы, которая подлежит отладке:
`td имя_исполняемого_модуля`

Кстати, сам файл отладчика `td.exe` логично также поместить в наш рабочий каталог WORK. Изначально файлы отладчика находятся в каталоге BIN пакета TASM. Если все же `td.exe` и исполняемый модуль при запуске будут находиться в разных каталогах, то в командной строке необходимо указать путь к этому модулю, например:

```
td c:\tasm\work\имя_модуля.exe
```

При правильном выполнении перечисленных выше действий откроется окно отладчика TD под названием `Module`.

В этом окне вы видите исходный текст программы `prg_3_1.asm`. Как он здесь оказался, ведь мы для программы `td.exe` указали только имя исполняемого модуля? Это как раз и есть результат действия опций /zi и /v для `tasm` и `tlink`, соответственно. Их применение позволило сохранить информацию об использовавшихся в коде на ассемблере символьических именах. Ради полноты эксперимента вы

можете получить исполняемый модуль без задания этих опций. Проанализируйте результат. Но вернемся к окну **Module**. Здесь вы видите так называемый *курсор выполнения* (в виде треугольника). Он указывает на первую команду, подлежащую выполнению. Этой команде предшествует имя метки (в нашем случае роль метки выполняет имя процедуры). Это так называемая *точка входа* в программу. Если вы внимательно посмотрите конец исходного текста программы, то увидите, что это же имя записано в качестве операнда в заключительной директиве **END**. Это единственный способ сообщить загрузчику ОС о том, где в исходном тексте программы расположена точка входа в нее. В более сложных программах обычно вначале могут идти описания процедур, макрокоманд, и в этом случае без такого явного указания на первую исполняемую команду вам не обойтись.

Основную часть экрана отладчика обычно занимают одно или несколько окон. В каждый момент времени активным может быть только одно из них. Активизация любого окна производится щелчком мышью в любой видимой точке окна. Управление работой отладчика ведется с помощью системы меню. Имеются два типа таких меню:

- **глобальное меню** — находится в верхней части экрана и доступно постоянно. Вызов меню осуществляется нажатием клавиши F10, после чего следует выбрать нужный пункт этого меню;
- **локальное меню** — для каждого окна отладчика можно вызвать его собственное меню, которое учитывает особенности этого окна. Вызвать данное меню можно, щелкнув в окне правой кнопкой мыши (либо сделав активным окно и нажав клавиши Alt+F10).

Теперь можно проверить правильность функционирования нашей программы.

Специфика программ на ассемблере в том, что делать выводы о правильности их функционирования можно, только отслеживая работу на уровне микропроцессора. При этом нас интересует прежде всего то, как программа использует микропроцессор и изменяет состояние его ресурсов и компьютера в целом.

Запустить программу на выполнение в отладчике можно в одном из четырех режимов:

- режим безусловного выполнения;
- выполнение по шагам;
- выполнение до текущего положения курсора;
- выполнение с установкой точек прерывания.

Рассмотрим эти режимы подробнее.

Режим безусловного выполнения целесообразно применять, когда требуется посмотреть на общее поведение программы. Для запуска программы в этом режиме необходимо нажать клавишу F9. В точках, где необходимо ввести данные, отладчик, в соответствии с логикой работы применяемого средства ввода, будет осуществлять определенные действия. Аналогичные действия отладчик выполняет для вывода данных. Для просмотра или ввода этой информации можно открыть окно *пользователя* (**Window ▶ User screen**)¹, например, нажав клавиши Alt+F5.

¹ В скобках указывается последовательность обращения к меню для выполнения необходимой операции.

Если работа программы удовлетворяет вас, то на этом можно и закончить. В случае если возникают какие-то проблемы, или если нужно более детально изучить работу программы, применяются три следующих режима отладки.

Выполнение по шагам применяется для детального изучения работы программы. В этом режиме вы можете выполнить программу по командам. При этом можно наблюдать результат исполнения каждой команды. Для активизации этого режима нужно нажать клавиши F7 (Run ▶ Trace into) или F8 (Run ▶ Step over). Обе эти клавиши активизируют пошаговый режим; отличие их проявляется в том случае, когда в потоке команд встречаются команды перехода в процедуру или на прерывание. При использовании клавиши F7 отладчик осуществит переход на процедуру или прерывание и выполнит их по шагам. Если же используется клавиша F8, то вызов процедуры или прерывания отрабатывается как одна обычная команда и управление передается следующей команде программы. Здесь нужно отметить, что кроме окна *Module* при работе в этом режиме полезно использовать окно *CPU*, вызвать которое можно через глобальное меню командой *View ▶ CPU*.

Окно *CPU* отражает состояние микропроцессора и состоит из 5 подчиненных окон:

- окно с исходной программой в дизассемблированном виде. Это та же самая программа, что и в окне *Module*, но уже в машинном виде. Пошаговую отладку можно производить прямо в этом окне; строка с текущей командой подсвечивается;
- *Registers* — окно регистров микропроцессора, отражающего текущее содержимое регистров. Заметьте, что по умолчанию отображаются регистры только i8086. Для того чтобы воспользоваться всеми регистрами i486 или Pentium, нужно задать режим их отображения. Для этого щелкните правой кнопкой мыши в области подокна регистров для вызова локального меню и выберите в нем команду *Registers 32-bit – Yes*;
- окно флагов, которое отражает текущее состояние флагов микропроцессора в соответствии с их мнемоническими называниями;
- окно стека *Stack*, отражающего содержимое памяти, выделенной для стека. Адрес области стека определяется содержимым регистров SS и SP;
- окно дампа оперативной памяти *Dump*, отражающее содержимое области памяти по адресу, который формируется из компонентов, указанных в левой части окна. В окне можно увидеть содержимое произвольной области памяти. Для этого нужно в локальном меню, вызываемом по щелчку правой кнопки мыши, выбрать нужную команду.

Некоторые из этих подчиненных окон можно вывести на экран отдельно. Все-таки удобнее работать с исходным текстом в окне *Module*, чем с его дизассемблированным вариантом в окне *CPU*. Но в то же время нужно отслеживать и состояние микропроцессора, используя информацию из подокон окна *CPU*. Совместить возможности окон *Module* и *CPU* можно посредством пункта глобального меню *View*, в котором необходимо выбрать нужные имена подчиненных окон *CPU*.

Выполнение до текущего положения курсора позволяет выполнить программу по шагам, начиная с произвольного места программы. Этот режим целесообразно использовать в том случае, если вас интересует только правильность функционирования некоторого участка программы. Для активизации этого режима необ-

ходимо установить курсор на нужную строку программы и нажать клавишу F4. Программа начнет выполнение и остановится на отмеченной команде, не выполнив ее. Далее вы можете использовать при необходимости пошаговый режим.

Выполнение с установкой точек прерывания позволяет выполнить программу с остановкой ее в строго определенных *точках прерывания* (breakpoints). Перед выполнением программы необходимо установить эти точки в программе, для чего следует перейти в нужную строку и нажать клавишу F2. Выбранные строки подсвечиваются другим цветом. Установленные ранее точки прерывания можно убрать — для этого нужно повторно выбрать нужные строки и нажать клавишу F2. После установки точек прерывания программа запускается на выполнение клавишей F9. На первой точке прерывания программа останавливается. Теперь можно посмотреть состояние микропроцессора и памяти, а затем продолжить выполнение программы. Сделать это можно или в пошаговом режиме, или выполнив программу до следующей точки прерывания.

Прервать выполнение программы в любом из этих режимов можно, нажав Ctrl+F2.

Мы описали все основные моменты, связанные с отладкой программы, начиная с ее загрузки в отладчик и заканчивая способами исследования ее работы. Теперь вам нужно немного попрактиковаться, запуская программу prg_3_1.exe в различных режимах отладчика или используя комбинацию этих режимов. Когда вы почувствуете, что освоились, переходите к следующему уроку.

Утилита MAKE

Процесс разработки программ с использованием пакетов TASM и MASM предполагает, что пользователь интенсивно работает с командной строкой. При этом он должен помнить не только последовательность запуска различных программ, формирующих исполняемый модуль, но и задаваемые при этом параметры этих программ. Если проект состоит из большого количества файлов, это довольно утомительно и может вызвать у программиста желание выбрать другое программное средство для реализации своего проекта. ›

Как решить такую проблему? Часто программист находит выход в использовании командных файлов (типа файлов .bat для MS-DOS). Но этот способ имеет существенный недостаток. Предположим, что исполняемый файл собирается компоновщиком из большого количества объектных файлов. В процессе отладки программист редко работает сразу с несколькими исходными файлами. Обычно изменяется один из исходных файлов, после чего для него транслятором создается объектный модуль, и далее компоновщик создает из всех объектных файлов единый исполняемый модуль. После этого программист запускает исполняемый модуль на выполнение и тестирует его. В ходе тестирования делаются выводы о том, насколько удовлетворяет программиста работа исполняемого модуля. Если возникают какие-то проблемы, то процесс внесения изменений циклически повторяется. Для сложных программ количество таких циклов может исчисляться десятками. И каждый раз нужно внимательно следить за правильностью задания параметров утилит пакета транслятора, формирующих исполняемый модуль. При

этом реально изменения вносятся только в один исходный модуль, а для построения исполняемого модуля к работе привлекаются все остальные модули проекта. Теоретически можно попытаться автоматизировать этот процесс с использованием командного файла. Но сразу оговоримся, что этот способ хорош только для самых простых случаев. Интерпретатор команд операционной системы, обеспечивающий работу командного файла, работает достаточно «бездумно», в частности он не имеет средств для «интеллектуального» отслеживания временных зависимостей между отдельными исходными файлами проекта. При использовании командного файла вам придется транслировать все файлы проекта вместо одного действительно измененного. Иначе, придется каждый раз вносить корректиды в текст командного файла, но тогда что же это за автоматизация? Очевидно, что все это ведет к потерям рабочего времени и затягивает процесс разработки проекта. С тем чтобы облегчить жизнь программиста в подобных случаях, существуют специальные программы, называемые *менеджерами проекта*.

В пакете TASM такой программой является утилита make.exe (в пакете MASM – nmake.exe). Эти утилиты работают со специально оформленными файлами, называемыми файлами описания, или make-файлами (далее make-файл). В make-файлах задаются отношения между файлами проекта и действия над этими файлами, которые выполняются в зависимости от возникновения определенных условий. Важно понимать, что файлы, между которыми задаются эти отношения, могут быть самые разные, например, текстовые, графические, исполняемые и т. д. Главный принцип, положенный в основу работы утилиты make.exe, заключается в анализе времени изменения или создания файлов.

Подробное описание утилиты make.exe вы найдете на диске в каталоге, соответствующем этому уроку.

Подведем некоторые итоги:

- Наиболее развитым и удобным для использования ассемблерным пакетом является транслятор фирмы Borland – TASM. Большое количество опций и директив управления листингом программы на ассемблере позволяют задать достаточно гибкую обработку исходного кода транслируемой программы.
- Процесс создания программ на ассемблере напоминает процесс создания программ на большинстве языков высокого уровня. Последние версии транслятора фирмы Borland, начиная с TASM 3.0, поддерживают все основные существующие на настоящий момент времени технологии программирования, включая объектно-ориентированную.
- Специфика разработки программы на ассемблере в том, что программист должен уделять внимание не только и не столько особенностям моделирования предметной области, сколько тому, как при этом наиболее эффективно и корректно использовать ресурсы микропроцессора.
- В результате работы транслятора создается листинг программы, содержащий разнообразную информацию о программе: объектный код, сообщения о син-

таксических ошибках, таблицу символов и т. д. Имея небольшой опыт, из листинга можно извлечь массу полезной информации.

- Целями трансляции программы являются обнаружение синтаксических ошибок и генерация объектного модуля. После того как получен корректный объектный модуль, программу необходимо скомпоновать. Для этого применяется утилита TLINK, входящая в состав пакета TASM. Одно из основных ее назначений — разрешение внешних ссылок. Если наша целевая программа состоит из нескольких отдельно оттранслированных модулей и в них есть взаимные ссылки на переменные или модули, то TLINK разрешает их, формируя тем самым правильные перемещаемые адреса.
- Результатом работы утилиты TLINK является исполняемый (загрузочный) модуль, имеющий расширение .exe. Его уже можно запускать на выполнение с надеждой, что он правильно выполнит задуманные вами действия. Так, возможно, и будет, когда у вас появится достаточно опыта и знаний. Пока же чаще всего вы будете сталкиваться с ситуацией, в которой машина погружается в глубокую задумчивость, выдает на экран какую-то бессмыслицу или вовсе перезагружается. Все это говорит о том, что вам предстоит долгая работа по поиску самых вредных ошибок — логических. К счастью, в вашем распоряжении есть специальный вид программного обеспечения — отладчики. Постарайтесь уделить им достойное внимание, и вы увидите, что большинство ваших программных проблем будет снято.



5

УРОК

Структура программы на ассемблере

-
- Структура программы на ассемблере
 - Стандартные директивы сегментации
 - Упрощенные директивы сегментации
 - Представление простых типов данных
-

На предыдущих уроках мы подробно разобрались с тем, что представляет собой микропроцессор изнутри, каковы принципы его работы и какая часть его архитектуры оставлена программируемой. Мы даже разработали одну полезную программу. Настало время подробно разобраться с самим языком ассемблера и с правилами оформления программ на нем.

Коротко вспомним информацию об ассемблере, которой мы обладаем на настоящий момент:

- ассемблер является символическим аналогом машинного языка. По этой причине программа, написанная на ассемблере, должна отражать все особенности архитектуры микропроцессора: организацию памяти, способы адресации операндов, правила использования регистров и т. д. Из-за необходимости учета подобных особенностей ассемблер уникален для каждого типа микропроцессоров;
- программа на ассемблере представляет собой совокупность блоков памяти, называемых сегментами памяти. Программа может состоять из одного или нескольких таких блоков-сегментов. Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы;
- предложения ассемблера бывают четырех типов:
 - *команды*, или инструкции, представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд микропроцессора;
 - *макрокоманды* — оформленные определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями;
 - *директивы*, являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении;
 - *строки комментариев*, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

Синтаксис ассемблера

Предложения, составляющие программу, могут представлять собой синтаксическую конструкцию, соответствующую команде, макрокоманде, директиве или комментарию. Для того, чтобы транслятор ассемблера мог распознать их, они должны формироваться по определенным синтаксическим правилам. Нам, конечно, было бы неплохо их знать. Лучше всего использовать формальное описание синтаксиса языка, наподобие правил грамматики. Наиболее распространенные способы подобного описания языка программирования — синтаксические диаграммы и расширенные формы Бэкуса-Наура. Для практического использования более удобны синтаксические диаграммы. К примеру, синтаксис предложений ассемблера можно описать с помощью синтаксических диаграмм, показанных на следующих рисунках.

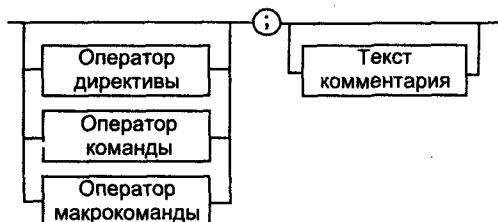


Рис. 5.1. Формат предложения ассемблера



Рис. 5.2. Формат директив

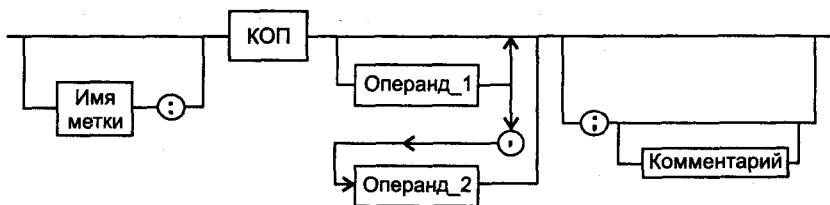


Рис. 5.3. Формат команд и макрокоманд

На этих рисунках:

- **имя метки** — идентификатор, значением которого является адрес первого байта того предложения исходного текста программы, которое он обозначает;
- **имя** — идентификатор, отличающий данную директиву от других одноименных директив. В результате обработки ассемблером определенной директивы этому имени могут быть присвоены определенные характеристики;

- **код операции** (КОП) и **директива** – это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора;
- **операнды** – части команды, макрокоманды или директивы ассемблера, обозначающие объекты, над которыми производятся действия. Операнды ассемблера описываются выражениями с числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.

Как использовать синтаксические диаграммы? Очень просто: для этого нужно всего лишь найти и затем пройти путь от входа диаграммы (слева) к ее выходу (направо). Если такой путь существует, то предложение или конструкция синтаксически правильны. Если такого пути нет, значит, эту конструкцию компилятор не примет. При работе с синтаксическими диаграммами обращайте внимание на направление обхода, указываемое стрелками, так как среди путей могут быть и такие, по которым можно идти справа налево. По сути, синтаксические диаграммы отражают логику работы транслятора при разборе входных предложений программы. В нашей книге синтаксические диаграммы будут помогать в сложных случаях описывать синтаксис конструкций ассемблера.

Допустимыми символами при написании текста программ являются:

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

Предложения ассемблера формируются из лексем, представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора. Лексемами являются:

- **идентификаторы** – последовательности допустимых символов, использующиеся для обозначения таких объектов программы, как коды операций, имена переменных и названия меток. Правило записи идентификаторов заключается в следующем. Идентификатор может состоять из одного или нескольких символов. В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки – _, ?, \$, @. Идентификатор не может начинаться символом цифры. Длина идентификатора может быть до 255 символов, хотя транслятор воспринимает лишь первые 32, а остальные игнорирует. Регулировать длину возможных идентификаторов можно с использованием опции командной строки **mv** (подробнее см. Справочник). Кроме этого, существует возможность указать транслятору на то, чтобы он различал прописные и строчные буквы либо игнорировал их различие (что и делается по умолчанию). Для этого применяются опции командной строки **/mu**, **/m1**, **/mx**;
- **цепочки символов** – последовательности символов, заключенные в одинарные или двойные кавычки;
- **целые числа** в одной из следующих систем счисления: двоичной, десятичной, шестнадцатеричной. Отождествление чисел при записи их в программах на ассемблере производится по определенным правилам. Для шестнадцатерич-

ных чисел эти правила были рассмотрены на уроке 3. Десятичные числа не требуют для своего отождествления указания каких-либо дополнительных символов. Для отождествления в исходном тексте программы двоичных чисел необходимо после записи нулей и единиц, входящих в их состав, поставить латинское «b». К примеру 10010101b.

Таким образом, мы разобрались с тем, как конструируются предложения программы ассемблера. Но это — лишь самый поверхностный взгляд. Практически каждое предложение содержит описание объекта, над которым или при помощи которого выполняется некоторое действие. Эти объекты называются операндами. Их можно определить так: операнды — это объекты (некоторые значения, регистры или ячейки памяти), на которые действуют инструкции или директивы, либо это объекты, которые определяют или уточняют действие инструкций или директив.

Операнды могут комбинироваться с арифметическими, логическими, побитовыми и атрибутивными операторами для расчета некоторого значения или определения ячейки памяти, на которую будет воздействовать данная команда или директива.

Рассмотрим классификацию операндов, поддерживаемых транслятором ассемблера.

○ *Постоянные, или непосредственные операнды*, — число, строка, имя или выражение, имеющие некоторое фиксированное значение. Имя не должно быть перемещаемым, то есть зависеть от адреса загрузки программы в память. К примеру, оно может быть определено операторами equ или =:

```
num    equ   5
imd = num-2
        mov   a1,num ;эквивалентно mov a1,5 ;здесь непосредственный
                  операнд
        add   [si],imd ; imd=3 – непосредственный операнд
        mov   a1,5       ;5 – непосредственный операнд
```

Это первый рассматриваемый нами фрагмент программы, содержащий машинные команды (не считая листинга 3.1, который преследовал другие цели). В свое время все они будут подробно описаны, пока же мы по мере необходимости будем пояснять их в комментариях. Для более подробной информации вы всегда можете обратиться к Справочнику, где приведены сведения по всем машинным командам. В данном фрагменте определяются две константы, которые затем используются в качестве непосредственных операндов в командах пересылки mov и сложения add.

○ *Адресные операнды* — задают физическое расположение операнда в памяти с помощью указания двух составляющих адреса: сегмента и смещения (рис. 5.4).

К примеру:

```
mov   ax,0000h
mov   ds,ax
mov   ax,ds:0000h ;записать слово в ax из области памяти
                  ;по физическому адресу 0000:0000
```

Здесь третья команда mov имеет адресный операнд.

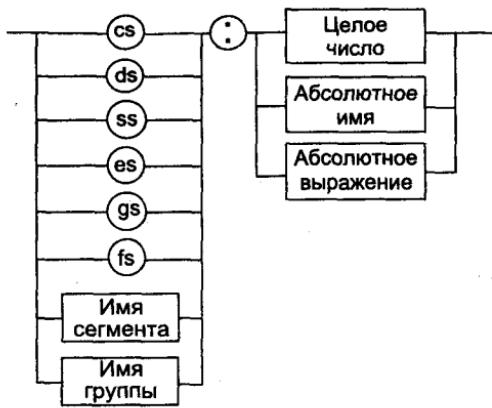


Рис. 5.4. Синтаксис описания адресных операндов

- **Перемещаемые операнды** – любые символьные имена, представляющие некоторые адреса памяти. Эти адреса могут обозначать местоположение в памяти некоторой инструкции (если operand – метка) или данных (если operand – имя области памяти в сегменте данных). Перемещаемые операнды отличаются от адресных тем, что они не привязаны к конкретному адресу физической памяти. Сегментная составляющая адреса перемещаемого операнда неизвестна и будет определена после загрузки программы в память для выполнения. К примеру:

```

data segment
mas_w dw 25 dup (0)
...
code segment
...
    lea si,mas_w ;mas_w – перемещаемый операнд

```

В этом фрагменте `mas_w` – символьное имя, значением которого является начальный адрес области памяти размером 25 слов. Полный физический адрес этой области памяти будет известен только после загрузки программы в память для выполнения.

- **Счетчик адреса** – специфический вид операнда. Он обозначается знаком `$`. Специфика этого операнда в том, что когда транслятор ассемблера встречает в исходной программе этот символ, то он подставляет вместо него текущее значение счетчика адреса. Значение счетчика адреса, или как его иногда называют *счетчика размещения*, представляет собой смещение текущей машинной команды относительно начала сегмента кода. Мы упоминали о счетчике адреса на уроке 4, когда обсуждали понятие листинга на примере программы урока 3 (`prg_3_1.asm`). В формате листинга счетчику адреса соответствует вторая или третья колонка (в зависимости от того, присутствует или нет в листинге колонка с уровнем вложенности). Как видите из этого примера листинга, при обработке транслятором очередной команды ассемблера счетчик адреса увеличивается на длину сформированной машинной команды. Важно правильно понимать этот момент. К примеру, обработка директив ассемблера не влечет за собой изменения счетчика. Подумайте, почему? Директивы, в отличие

от команд ассемблера, — это лишь указания транслятору на выполнение определенных действий по формированию машинного представления программы, и для них транслятором не генерируется никаких конструкций в памяти. В качестве примера использования в команде значения счетчика адреса можно привести следующий фрагмент:

```
jmp    $+3    ;безусловный переход на команду mov  
cld    ;длина команды cld составляет 1 байт  
mov a1,1
```

При использовании подобного выражения для перехода не забывайте о длине самой команды, в которой это выражение используется, так как значение счетчика адреса соответствует смещению в сегменте команд данной, а не следующей за ней команды. В нашем примере команда `jmp` занимает 2 байта. Но будьте осторожны, длина этой и других команд зависит от того, какие в ней используются операнды. Команда с регистровыми операндами будет короче команды, один из операндов которой расположен в памяти. В большинстве случаев эту информацию можно получить, зная формат машинной команды (см. урок 6) и анализируя колонку листинга с объектным кодом команды.

- *Регистровый operand* — это просто имя регистра. В программе на ассемблере можно использовать имена всех регистров общего назначения и большинства системных регистров.

```
mov    a1,4    ;константу 4 заносим в регистр a1  
mov    d1,pass+4  ;байт по адресу pass+4 в регистр d1  
add    a1,d1    ;команда с регистровыми операндами
```

- *Базовый и индексный operandы*. Этот тип operandов используется для реализации косвенной базовой, косвенной индексной адресации или их комбинаций и расширений. Адресация будет рассмотрена на следующем уроке.
- *Структурные operandы* используются для доступа к конкретному элементу сложного типа данных, называемого *структурой*. Мы подробно разберемся со структурами на уроке 12.
- *Записи* (аналогично структурному типу) используются для доступа к битово-му полю некоторой записи (см. урок 12).

Операнды являются элементарными компонентами, из которых формируется часть машинной команды, обозначающая объекты, над которыми выполняется операция. В более общем случае operandы могут входить как составные части в более сложные образования, называемые *выражениями*. Выражения представляют собой комбинации operandов и *операторов*, рассматриваемые как единое целое. Результатом вычисления выражения может быть адрес некоторой ячейки памяти или некоторое константное (абсолютное) значение.

Перечислим возможные типы операторов ассемблера и синтаксические правила формирования выражений ассемблера. Заметим только, что как и в языках высокого уровня, выполнение операторов ассемблера при вычислении выражений осуществляется в соответствии с их приоритетами (табл. 5.1). Операции с одинаковыми приоритетами выполняются последовательно слева направо. Изменение порядка выполнения возможно путем расстановки круглых скобок, которые имеют наивысший приоритет.

Таблица 5.1. Операторы и их приоритет

| Оператор | Приоритет |
|--|-----------|
| length, size, width, mask, (,), [,], <, > | 1 |
| : | 2 |
| ptr, offset, seg, type, this | 3 |
| high, low | 4 |
| +, - (унарные) | 5 |
| *, /, mod, shl, shr | 6 |
| +, -, (бинарные) | 7 |
| eq, ne, lt, le, gt, ge | 8 |
| not | 9 |
| and | 10 |
| or, xor | 11 |
| short, type | 12 |
| | 13 |

Дадим краткую характеристику операторов.

- **Арифметические операторы.** К ним относятся унарные операторы «+» и «-», бинарные «+» и «-», операторы умножения «*», целочисленного деления «/», получения остатка от деления «mod» (рис. 5.5). Эти операторы расположены на уровнях приоритета 6, 7, 8 в табл. 5.1.

Например:

```
tab_size    equ   50      ;размер массива в байтах
size_el     equ   2       ;размер элементов
;
;вычисляется число элементов массива и заносится в регистр cx
        mov  cx,tab_size / size_el    ;оператор "/"
```

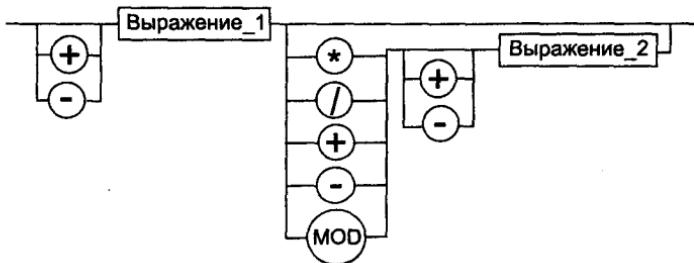


Рис. 5.5. Синтаксис арифметических операторов

- **Операторы сдвига** выполняют сдвиг выражения на указанное количество разрядов (рис. 5.6).

Например:

```
mask_b equ 10111011
-
        mov  al,mask_b shr 3      ;al=00010111
```

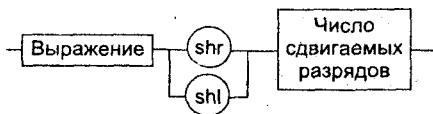


Рис. 5.6. Синтаксис операторов сдвига

- О **Операторы сравнения** (возвращают значение «истина» или «ложь») предназначены для формирования логических выражений (см. рис. 5.7 и табл. 5.2). Логическое значение «истина» соответствует логической единице, а «ложь» — логическому нулю. Логическая единица — значение, все биты которого равны 1. Соответственно, логический нуль — значение, все биты которого равны 0. Например:

```
tab_size equ 30 ;размер таблицы
...
    mov al,tab_size ge 50 ;загрузка размера таблицы в al
    cmp al,0 ;если tab_size < 50, то
    je m1 ;переход на m1
```

m1: ...

В этом примере, если значение `tab_size` больше или равно 50, то результат в `al` равен `0ffh`, а если `tab_size` меньше 50, то `al` равно `00h`. Команда `cmp` сравнивает значение `al` с нулем и устанавливает соответствующие флаги в `flags/eflags`. Команда `je` на основе анализа этих флагов передает или не передает управление на метку `m1`.

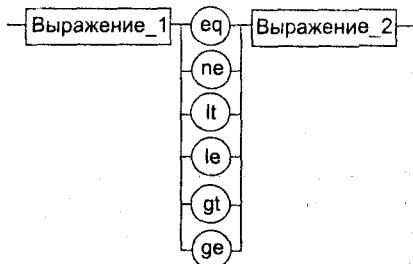


Рис. 5.7. Синтаксис операторов сравнения

Таблица 5.2. Операторы сравнения

| Оператор | Значение (см. рис. 5.7) |
|----------|---|
| eq | ИСТИНА, если выражение_1 равно выражение_2 |
| ne | ИСТИНА, если выражение_1 не равно выражение_2 |
| lt | ИСТИНА, если выражение_1 меньше выражение_2 |
| le | ИСТИНА, если выражение_1 меньше или равно выражение_2 |
| gt | ИСТИНА, если выражение_1 больше выражение_2 |
| ge | ИСТИНА, если выражение_1 больше или равно выражение_2 |

- Логические операторы выполняют над выражениями побитовые операции (рис. 5.8). Выражения должны быть абсолютными, то есть такими, численное значение которых может быть вычислено транслятором.

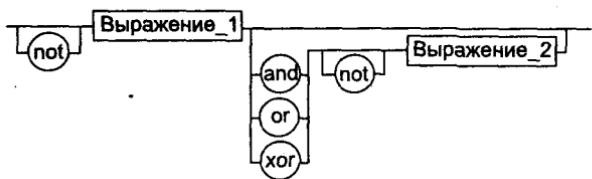


Рис. 5.8. Синтаксис логических операторов

Например:

Более подробно о правилах, в соответствии с которыми вычисляется результат логических операций, мы поговорим на уроке 9.

- *Индексный оператор []*. Не удивляйтесь, но скобки тоже являются оператором, и транслятор их наличие воспринимает как указание сложить значение выражение_1 за этими скобками со значением выражение_2, заключенным в скобки (рис. 5.9).

Например:

mov ax,mas[si] ;пересылка слова по адресу mas+(si) в регистр ax

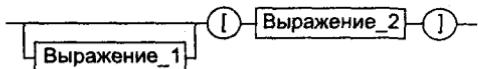


Рис. 5.9. Синтаксис индексного оператора

Заметим, что в литературе принято следующее обозначение: когда в тексте речь идет о содержимом регистра, то его название берут в круглые скобки. Мы также будем придерживаться этого обозначения. К примеру, в нашем случае запись в комментариях последнего фрагмента программы `mas + (si)` означает вычисление выражения: значение смещения символьического имени `mas` плюс содержимое регистра `si`.

- Оператор переопределения `using ptr` применяется для переопределения или уточнения типа метки или переменной, определяемых выражением (рис. 5.10). Тип может принимать одно из следующих значений: `byte`, `word`, `dword`, `qword`, `tbyte`, `near`, `far`. Что означают эти значения, вы узнаете далее на этом уроке.

Например:

d wrd dd 0

...
mov al,byte ptr d_wrd+1 ;пересылка второго байта из двойного слова

Поясним этот фрагмент программы. Переменная `d_wrd` имеет тип двойного слова. Что делать, если возникнет необходимость обращения не ко всему значению переменной, а только к одному из входящих в нее байтов (например, ко второму)? Если попытаться сделать это командой `mov al,d_wrd+1`, то транслятор выдаст сообщение о несовпадении типов операндов. Оператор `ptr` позволяет непосредственно в команде переопределить тип и выполнить команду.

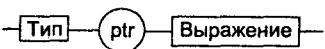


Рис. 5.10. Синтаксис оператора переопределения типа

- *Оператор переопределения сегмента* : (двоеточие) заставляет вычислять физический адрес относительно конкретно задаваемой сегментной составляющей: «имя сегментного регистра», «имя сегмента» из соответствующей директивы `SEGMENT` или «имя группы» (рис. 5.11).

Этот момент очень важен, поэтому поясним его подробнее. При обсуждении сегментации мы говорили о том, что микропроцессор на аппаратном уровне поддерживает три типа сегментов — кода, стека и данных. В чем заключается такая аппаратная поддержка? К примеру, для выборки на выполнение очередной команды микропроцессор должен обязательно посмотреть содержимое сегментного регистра `cs` и только его. А в этом регистре, как мы знаем, содержится (пока еще не сдвинутый) физический адрес начала сегмента команд. Для получения адреса конкретной команды микропроцессору остается умножить содержимое `cs` на 16 (что означает сдвиг на четыре разряда) и сложить полученное 20-битное значение с 16-битным содержимым регистра `ip`. Примерно то же самое происходит и тогда, когда микропроцессор обрабатывает операнды в машинной команде. Если он видит, что операнд — это адрес (эффективный адрес, который является только частью физического адреса), то он знает, в каком сегменте его искать, — по умолчанию это сегмент, адрес начала которого записан в сегментном регистре `ds`.

А что же с сегментом стека? Посмотрите урок 2 там, где мы описывали назначение регистров общего назначения. В контексте нашего рассмотрения нас интересуют регистры `sp` и `bp`. Если микропроцессор видит в качестве операнда (или его части, если операнд — выражение) один из этих регистров, то по умолчанию он формирует физический адрес операнда, используя в качестве его сегментной составляющей содержимое регистра `ss`. Что подразумевает термин «по умолчанию»? Вспомните «рефлексы», о которых мы говорили на уроке 1. Это набор микропрограмм в блоке микропрограммного управления, каждая из которых выполняет одну из команд в системе машинных команд микропроцессора. Каждая микропрограмма работает по своему алгоритму. Изменить его, конечно же, нельзя, но можно чуть-чуть подкорректировать. Делается это с помощью необязательного поля префикса машинной команды (см. формат команд в уроке 2). Если мы согласны с тем, как работает команда, то это поле отсутствует. Если же мы хотим внести поправку (если, конечно, она допустима для конкретной команды) в алгоритм работы команды, то необходимо сформировать соответствующий префикс. Префикс представляет собой однобайтовую величину, численное значение

ние которой определяет ее назначение. Микропроцессор распознает по указанному значению, что этот байт является префиксом, и дальнейшая работа микропрограммы выполняется с учетом поступившего указания на корректировку ее работы. По ходу обсуждения материала книги мы познакомимся с большинством возможных префиксов. Сейчас нас интересует один из них — *префикс замены (переопределения) сегмента*. Его назначение состоит в том, чтобы указать микропроцессору (а по сути, микропрограмме) на то, что мы не хотим использовать сегмент по умолчанию. Возможности для подобного переопределения, конечно, ограничены. Сегмент команд переопределить нельзя, адрес очередной исполняемой команды однозначно определяется парой cs:ip. А вот сегменты стека и данных — можно. Для этого и предназначен оператор «:». Транслятор ассемблера, обрабатывая этот оператор, формирует соответствующий однобайтовый префикс замены сегмента. Например:

.code

```
jmp    met1  ;обход обязательен, иначе поле ind будет трактоваться
        ;как очередная команда
ind    db      5      ;описание поля данных в сегменте команд
met1:
...
mov    al,cs:ind  ;переопределение сегмента позволяет работать
        ;с данными, определенными внутри сегмента кода
```

Продолжим перечисление операторов.

- *Оператор именования типа структуры* . (точка) также заставляет транслятор производить определенные вычисления, если он встречается в выражении. Подробно мы обсудим этот оператор, когда введем понятие сложных типов данных на уроке 12.
- *Оператор получения сегментной составляющей адреса выражения* seg возвращает физический адрес сегмента для выражения (рис. 5.12), в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое симвлическое имя.

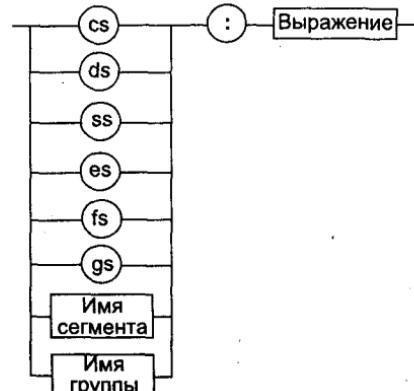


Рис. 5.11. Синтаксис оператора переопределения сегмента

Рис. 5.12. Синтаксис оператора получения сегментной составляющей

- О **Оператор получения смещения выражения offset** позволяет получить значение смещения выражения (рис. 5.13) в байтах относительно начала того сегмента, в котором выражение определено.

**Рис. 5.13.** Синтаксис оператора получения смещения

Например:

```

.data
pole dw 5
...
.code
...
    mov ax,seg pole
    mov es,ax
    mov dx,offset pole ;теперь в паре es:dx полный адрес pole

```

Директивы сегментации

В ходе предыдущего обсуждения мы выяснили все основные правила записи команд и operandов в программе на ассемблере. Открытым остался вопрос о том, как правильно оформить последовательность команд, чтобы транслятор мог их обработать, а микропроцессор — выполнить. На предыдущих уроках мы уже касались понятия сегмента. При рассмотрении архитектуры микропроцессора мы узнали, что он имеет шесть сегментных регистров, посредством которых может одновременно работать:

- с одним сегментом кода;
- одним сегментом стека;
- одним сегментом данных;
- тремя дополнительными сегментами данных.

Еще раз вспомним, что физически сегмент представляет собой область памяти, занятую командами и (или) данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Синтаксическое описание сегмента на ассемблере представляет собой конструкцию, изображенную на рис. 5.14.

Важно отметить, что функциональное назначение сегмента несколько шире, чем простое разбиение программы на блоки кода, данных и стека. Сегментация является частью более общего механизма, связанного с концепцией модульного программирования. Она предполагает унификацию оформления объектных модулей, создаваемых компилятором, в том числе с разных языков программирования. Это позволяет объединять программы, написанные на разных языках. Именно для

реализации различных вариантов такого объединения и предназначены операнды в директиве **SEGMENT**. Рассмотрим их подробнее:

○ Атрибут **выравнивания сегмента** (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в процессорах i80x86 выполняется быстрее. Допустимые значения этого атрибута следующие:

- **BYTE** — выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;
- **WORD** — сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);
- **DWORD** — сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита равны 0 (выравнивание на границу двойного слова);
- **PARA** — сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);
- **PAGE** — сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу страницы размером 256 байт);
- **MEMPAGE** — сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей страницы памяти размером 4 Кбайт);

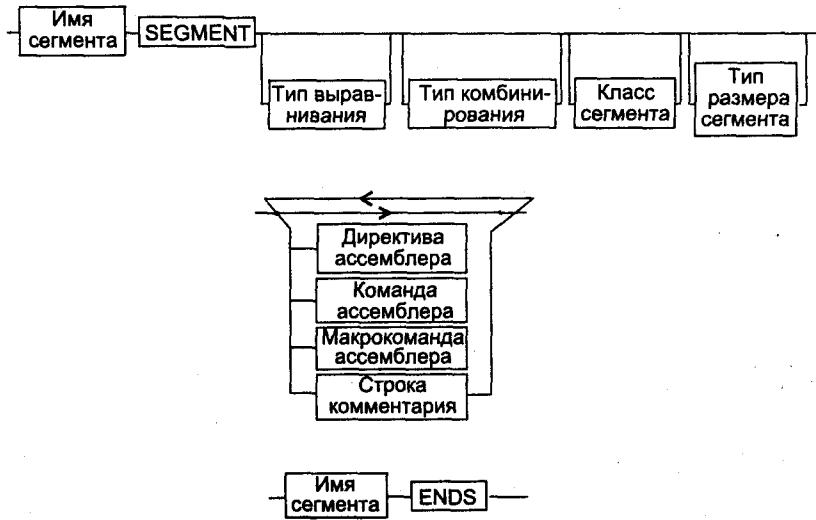


Рис. 5.14. Синтаксис описания сегмента

По умолчанию тип выравнивания имеет значение PARA.

○ Атрибут комбинирования сегментов (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. По умолчанию атрибут комбинирования принимает значение PRIVATE. Значениями атрибута комбинирования сегмента могут быть:

- PRIVATE — сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
- PUBLIC — заставляет компоновщик соединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;
- COMMON — располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
- AT xxxx — располагает сегмент по абсолютному адресу параграфа (*параграф* — объем памяти, кратный 16; потому последняя шестнадцатеричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением *xxx*. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;
- STACK — определение *сегмента стека*. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра *ss*. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC, за исключением того, что регистр *ss* является стандартным сегментным регистром для сегментов стека. Регистр *sp* устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр *ss* адрес сегмента (подобно тому, как это делается для регистра *ds*).

○ Атрибут класса сегмента (тип класса) — это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при сборке программы из сегментов нескольких модулей. Компоновщик объединяет вместе в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента). Типичным примером использования имени класса является объединение в группу всех сегментов кода программы (обычно для этого используется класс «code»). С помощью механизма типизации класса можно группировать также сегменты инициализированных и неинициализированных данных.

○ **Атрибут размера сегмента.** Для процессоров i80386 и выше сегменты могут быть 16- или 32-разрядными. Это влияет прежде всего на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:

- USE16 — это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;
- USE32 — сегмент будет 32-разрядным. При формировании физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Все сегменты сами по себе равноправны, так как директивы SEGMENT и ENDS не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом, для чего используют специальную директиву ASSUME, имеющую формат, показанный на рис. 5.15. Эта директива сообщает транслятору о том, какой сегмент к какому сегментному регистру привязан. В свою очередь, это позволит транслятору корректно связывать символические имена, определенные в сегментах. Привязка сегментов к сегментным регистрам осуществляется с помощью операндов этой директивы, в которых имя_сегмента должно быть именем сегмента, определенным в исходном тексте программы директивой SEGMENT или ключевым словом nothing. Если в качестве операнда используется только ключевое слово nothing, то предшествующие назначения сегментных регистров аннулируются, причем сразу для всех шести сегментных регистров. Но ключевое слово nothing можно использовать вместо аргумента имя сегмента; в этом случае будет выборочно разрываться связь между сегментом с именем имя сегмента и соответствующим сегментным регистром (см. рис. 5.15).

На уроке 3 мы рассматривали пример программы с директивами сегментации. Эти директивы изначально использовались для оформления программы в трансляторах MASM и TASM. Поэтому их называют стандартными директивами сегментации.

Для простых программ, содержащих по одному сегменту для кода, данных и стека, хотелось бы упростить их описание. Для этого в трансляторы MASM и TASM ввели возможность использования упрощенных директив сегментации. Но здесь возникла проблема, связанная с тем, что необходимо было как-то компенсировать невозможность напрямую управлять размещением и комбинированием сегментов. Для этого совместно с упрощенными директивами сегментации стали использовать директиву указания модели памяти MODEL, которая частично стала управлять размещением сегментов и выполнять функции директивы ASSUME (поэтому при использовании упрощенных директив сегментации директиву ASSUME можно не использовать). Эта директива связывает сегменты, которые, в случае использования упрощенных директив сегментации, имеют предопределенные имена с сегментными регистрами (хотя явно инициализировать ds все равно придется).

Используя упрощенные директивы сегментации для оформления программы из листинга 3.1, получим листинг 5.1.

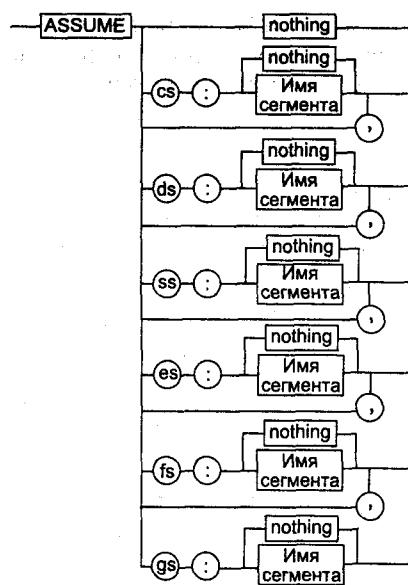


Рис. 5.15. Директива ASSUME

Листинг 5.1. Упрощенные директивы сегментации

```

;---Prg_3_1.asm---
masm           ;режим работы TASM: ideal или masm
model small   ;модель памяти
.data          ;сегмент данных
message db     "Введите две шестнадцатеричные цифры,$"
.stack         ;сегмент стека
db 256 dup (?) ;сегмент стека
.code          ;сегмент кода
main proc      ;начало процедуры main
    mov ax,@data ;заносим адрес сегмента данных в регистр ax
    mov ds,ax ;ax в ds
;далее текст программы идентичен тексту сегмента кода в листинге 3.1
    mov ax,4C00h ;пересылка 4C00h в регистр ax
    int 21h ;вызов прерывания с номером 21h
main endp       ;конец процедуры main
endmain        ;конец программы с точкой входа main

```

Синтаксис директивы MODEL показан на рис. 5.16:



Рис. 5.16. Синтаксис директивы MODEL

Обязательным параметром директивы **MODEL** является модель памяти. Этот параметр определяет модель сегментации памяти для программного модуля. Предполагается, что программный модуль может иметь только определенные типы сегментов, которые определяются так называемыми упрощенными директивами описания сегментов. Эти директивы приведены в табл. 5.3.

Наличие в некоторых директивах параметра [имя] говорит о том, что возможно определение нескольких сегментов этого типа. С другой стороны, наличие нескольких видов сегментов данных обусловлено требованием обеспечить совместимость с некоторыми компиляторами языков высокого уровня, которые создают разные сегменты данных для инициализированных и неинициализированных данных, а также констант.

Таблица 5.3. Упрощенные директивы определения сегмента

| Формат директивы (режим MASM) | Формат директивы (режим IDEAL) | Назначение |
|----------------------------------|-----------------------------------|---|
| .CODE [имя] | CODESEG [имя] | Начало или продолжение сегмента кода |
| .DATA | DATASEG | Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа neag ¹ |
| .CONST | CONST | Начало или продолжение сегмента постоянных данных (констант) модуля |
| .DATA? | UDATASEG | Начало или продолжение сегмента неинициализированных данных. Также используется для определения данных типа neag |
| .STACK [размер] | STACK [размер] | Начало или продолжение сегмента стека модуля. Параметр [размер] задает размер стека |
| .FARDATA [имя] | FARDATA [имя] | Начало или продолжение сегмента инициализированных данных типа far |
| .FARDATA? [имя] | UFARDATA [имя] | Начало или продолжение сегмента неинициализированных данных типа far |

При использовании директивы **MODEL** транслятор делает доступными несколько идентификаторов, к которым можно обращаться во время работы программы с тем, чтобы получить информацию о тех или иных характеристиках данной модели памяти (табл. 5.5). Перечислим эти идентификаторы и их значения (табл. 5.4).

¹ На этом занятии читатель довольно часто встречает служебные слова neag и far. Подробно они будут описаны при рассмотрении команд перехода и процедур. Пока же читатель должен представлять себе следующее. Слова neag и far определяют «удаленность» объекта от места использования. В качестве объектов могут выступать символические имена меток, переменных или процедур. Для доступа к ним, естественно, должен быть сформирован адрес. Так вот, если объект расположен «близко» (neag) и для доступа к нему нет необходимости менять значение соответствующего сегментного регистра, то говорят, что это объект типа neag; в противном случае — это объект типа far (дальний).

Таблица 5.4. Идентификаторы, создаваемые директивой MODEL

| Имя идентификатора | Значение переменной |
|---------------------------|--|
| @code | Физический адрес сегмента кода |
| @data | Физический адрес сегмента данных типа near |
| @fardata | Физический адрес сегмента данных типа far |
| @fardata? | Физический адрес сегмента неинициализированных данных типа far |
| @curseg | Физический адрес сегмента неинициализированных данных типа far |
| @stack | Физический адрес сегмента стека |

Если вы посмотрите на текст листинга 5.1, то увидите пример использования одного из этих идентификаторов. Это @data; с его помощью мы получили значение физического адреса сегмента данных нашей программы.

Теперь можно закончить обсуждение директивы MODEL. Операнды директивы MODEL используются для задания модели памяти, которая определяет набор сегментов программы, размеры сегментов данных и кода, способ связывания сегментов и сегментных регистров. В табл. 5.5 приведены некоторые значения параметров модели памяти директивы MODEL.

Таблица 5.5. Модели памяти

| Модель | Тип кода | Тип данных | Назначение модели |
|---------------|-----------------|-------------------|--|
| TINY | near | near | Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата .com |
| SMALL | near | near | Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на ассемблере |
| MEDIUM | far | near | Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления — типа far. Данные объединены в одной группе; все ссылки на них — типа near |
| COMPACT | near | far | Код в одном сегменте; ссылка на данные — типа far |
| LARGE | far | far | Код в нескольких сегментах, по одному на каждый объединяемый программный модуль |
| FLAT | near | near | Код и данные в одном 32-битном сегменте (плоская модель памяти) |

Параметр модификатор директивы MODEL позволяет уточнить некоторые особенности использования выбранной модели памяти (табл. 5.6).

Таблица 5.6. Модификаторы модели памяти

| Значение модификатора | Назначение |
|-----------------------|---|
| use16 | Сегменты выбранной модели используются как 16-битные (если соответствующей директивой указан процессор i80386 или i80486) |
| use32 | Сегменты выбранной модели используются как 32-битные (если соответствующей директивой указан процессор i80386 или i80486) |
| dos | Программа будет работать в MS-DOS |

Необязательные параметры язык и модификатор языка определяют некоторые особенности вызова процедур. Необходимость в использовании этих параметров появляется при написании и связывании программ на различных языках программирования. Этого вопроса мы еще коснемся на уроке 14 при обсуждении средств модульного программирования на ассемблере.

Описанные нами стандартные и упрощенные директивы сегментации не исключают друг друга. Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей. Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

Таким образом, в ходе предыдущего изложения мы разобрались со структурой программы на ассемблере. Мы выяснили, что программа разбивается на несколько сегментов, каждый из которых имеет свое функциональное назначение. Для такого разбиения нужно использовать директивы сегментации. TASM предоставляет два типа таких директив: стандартные и упрощенные. Упрощенные директивы сегментации мы и будем использовать в дальнейшем, если, конечно, у нас не возникнет необходимости применить стандартные директивы.

Простые типы данных ассемблера

Вторую часть урока мы посвятим описанию данных в программе на ассемблере. Любая программа предназначена для обработки некоторой информации, поэтому вопрос о том, как описать данные с использованием средств языка, обычно встает одним из первых. TASM предоставляет очень широкий набор средств описания и обработки данных, который вполне сравним с аналогичными средствами некоторых языков высокого уровня.

Начнем обсуждение с правил описания простых типов данных, которые являются базовыми для описания более сложных типов. Для описания простых типов данных в программе используются специальные директивы резервирования и инициализации данных, которые по сути являются указаниями транслятору на

выделение определенного объема памяти. Если проводить аналогию с языками высокого уровня, то директивы резервирования и инициализации данных являются определениями переменных. Машинного эквивалента этим директивам нет; просто транслятор, обрабатывая каждую такую директиву, выделяет необходимое количество байт памяти и при необходимости инициализирует эту область некоторым значением. Директивы резервирования и инициализации данных простых типов имеют формат, показанный на рис. 5.17.

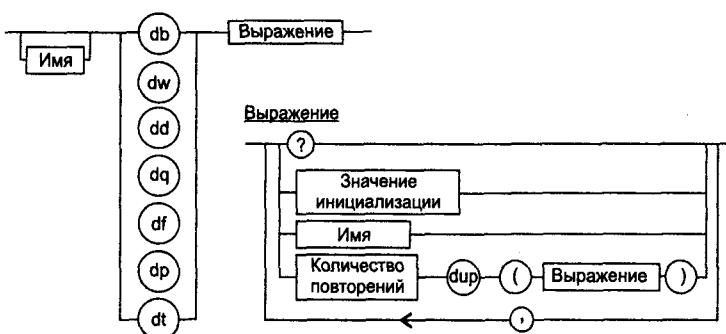


Рис. 5.17. Директивы описания данных простых типов

На рис. 5.17 использованы следующие обозначения.

- ? показывает, что содержимое поля не определено, то есть при задании директивы с таким значением выражения содержимое выделенного участка физической памяти изменяться не будет. Фактически создается неинициализированная переменная;
- значение инициализации — значение элемента данных, которое будет занесено в память после загрузки программы. Фактически создается инициализированная переменная, в качестве которой могут выступать константы, строки символов, константные и адресные выражения в зависимости от типа данных. Подробная информация приведена в Справочнике;
- выражение — итеративная конструкция с синтаксисом, описанным на рис. 5.17. Эта конструкция позволяет повторить последовательное занесение в физическую память выражения в скобках *n* раз.
- имя — некоторое символическое имя метки или ячейки памяти в сегменте данных, используемое в программе.

На рис. 5.17 представлены следующие поддерживаемые TASM директивы резервирования и инициализации данных:

- db — резервирование памяти для данных размером 1 байт;
- dw — резервирование памяти для данных размером 2 байта;
- dd — резервирование памяти для данных размером 4 байта;
- df — резервирование памяти для данных размером 6 байтов;
- dp — резервирование памяти для данных размером 6 байтов;
- dq — резервирование памяти для данных размером 8 байтов;
- dt — резервирование памяти для данных размером 10 байтов.

Очень важно уяснить себе порядок размещения данных в памяти. Он напрямую связан с логикой работы микропроцессора с данными. Микропроцессоры Intel требуют следования данных в памяти по принципу: младший байт по младшему адресу.

Для иллюстрации данного принципа рассмотрим листинг 5.2, в котором определим сегмент данных. В этом сегменте данных приведено несколько директив описания простых типов данных. Используя последовательность шагов, описанную на уроке 3, получим загрузочный модуль.

Листинг 5.2. Пример использования директив резервирования и инициализации данных

```
masm
model small
.stack 100h
.data
message db "Запустите эту программу в отладчике", '$'
perem_1 db 0ffh
perem_2 dw 3a7fh
perem_3 dd 0f54d567ah
mas db 10 dup (" ")
pole_1 db 5 dup (?)
adr dw perem_3
adr_full dd perem_3
fin db "Конец сегмента данных программы $"
.code
start:
    mov ax,@data
    mov ds,ax
    mov ah,09h
    mov dx,offset message
    int 21h
    mov ax,4c00h
    int 21h
end start
```

Теперь наша цель — посмотреть, как выглядит сегмент данных программы листинга 5.2 в памяти компьютера. Это даст нам возможность обсудить практическую реализацию обозначенного нами принципа размещения данных. Для этого запустим отладчик `td.exe`, входящий в комплект поставки TASM. Опишем этот процесс по шагам.

Введем код из листинга 5.2 в файл с названием `prg_5_2.asm`. Как мы договорились раньше, все манипуляции с файлом будем производить в директории `work`, где уже содержатся все необходимые для компиляции, компоновки и отладки файлы пакета TASM.

Запустим процесс трансляции файла следующей командой:

```
tasm.exe /zi prg_5_2.asm , , ,
```

После устранения синтаксических ошибок запустим процесс компоновки объектного файла:

```
tlink.exe /v prg_5_2.obj
```

Теперь можно производить отладку:

```
td prg_5_2.exe
```

Если все было сделано правильно, то в отладчике откроется окно **Module** с исходным текстом программы. Для того чтобы с помощью отладчика просмотреть область памяти, содержащую наш сегмент данных, необходимо открыть окно **Dump**. Это делается с помощью команды главного меню **View ▶ Dump**.

Но одного открытия окна недостаточно; нужно еще настроить его на адрес начала сегмента данных. Этот адрес должен содержаться в сегментном регистре **ds**, но, как сказано выше, перед началом выполнения программы адрес в **ds** не соответствует началу сегмента данных. Нужно перед первым обращением к любому символическому имени произвести загрузку действительного физического адреса сегмента данных. Обычно это действие не откладывают в долгий ящик и производят первыми двумя командами в сегменте кода. Действительный физический адрес сегмента данных извлекают как значение предопределенной переменной **@data**. В нашей программе эти действия выполняют команды:

```
mov ax,@data  
mov ds,ax
```

Для того чтобы посмотреть содержимое нашего сегмента данных, нужно остановить выполнение программы после этих двух команд. Это можно сделать, если перевести отладчик в пошаговый режим с помощью клавиш **F7** или **F8**. Нажмите два раза **F8**. Теперь можно открыть окно **Dump**.

В окне **Dump** вызовите контекстное меню, щелкнув правой кнопкой мыши. В появившемся контекстном меню выберите команду **Goto**. Появится диалоговое окно, в котором нужно ввести начальный адрес памяти, начиная с которого будет выводиться информация в окне **Dump**. Синтаксис задания этого адреса должен соответствовать синтаксису задания адресного операнда в программе на ассемблере (см. начало этого урока). Мы бы хотели увидеть содержимое памяти для нашего сегмента данных, начиная с его начала, поэтому введем **ds:0000**. Для удобства, если сегмент достаточно велик, это окно можно распахнуть на весь экран. Для этого нужно щелкнуть на символе «↑» в правом верхнем углу окна **Dump**. Вид экрана показан на рис. 5.18.

Обсудим рис. 5.18. На нем вы видите данные вашего сегмента в двух представлениях: шестнадцатеричном и символьном. Видно, что со смещением **0000** расположены символы, входящие в строку **message**. Она занимает 34 байта. После нее следует байт, имеющий в сегменте данных символическое имя **reget_1**, содержимое этого байта **offh**. Теперь обратите внимание на то, как размещены в памяти байты, входящие в слово, обозначенное символическим именем **reget_2**. Сначала следует байт со значением **7fh**, а затем со значением **3ah**. Как видите, в памяти действительно сначала расположен младший байт значения, а затем старший. Теперь посмотрите и самостоятельно проанализируйте размещение байтов для поля, обозначенного символическим именем **reget_3**. Оставшуюся часть сегмента данных вы можете также проанализировать самостоятельно. Остановимся лишь на двух специфических особенностях использования директив резервирования и

инициализации памяти. Речь идет о случае использования в поле операндов директив `dw` и `dd` символьического имени из поля имя этой или другой директивы резервирования и инициализации памяти. В нашем примере сегмента данных это директивы с именами `adr` и `adr_full`. Когда транслятор встречает директивы описания памяти с подобными operandами, то он формирует в памяти значения адресов тех переменных, чьи имена были указаны в качестве operandов. В зависимости от директивы, применяемой для получения такого адреса, формируется либо полный адрес (директива `dd`) в виде двух байтов сегментного адреса и двух байтов смещения, либо только смещение (директива `dw`). Найдите в дампе на рис. 5.18 поля, соответствующие именам `adr` и `adr_full`, и проанализируйте их содержимое.

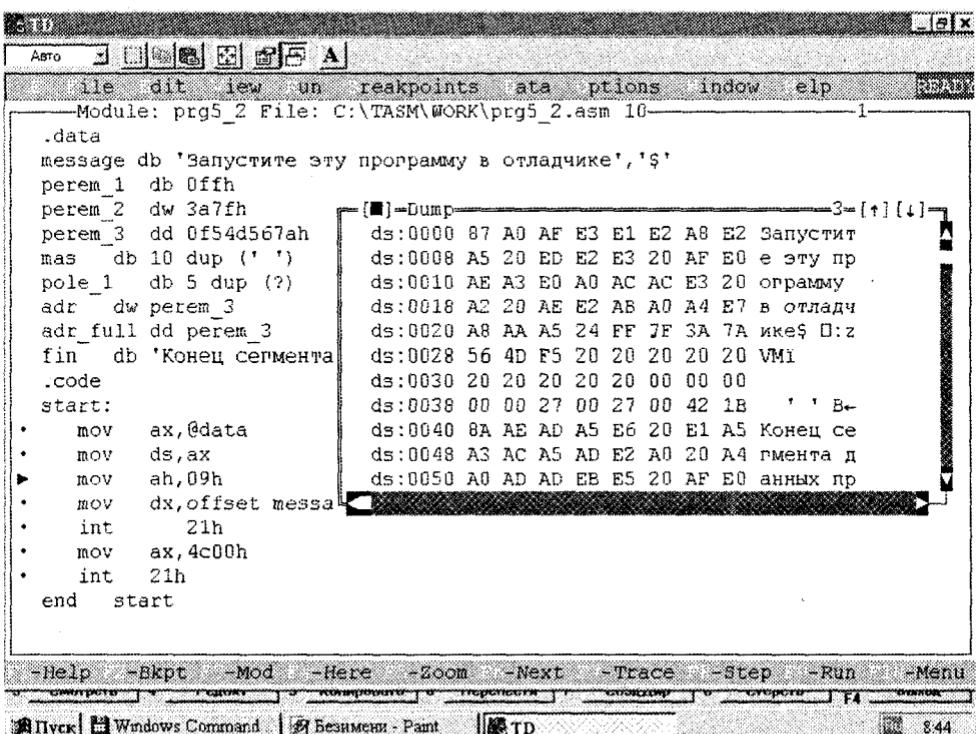


Рис. 5.18. Окно дампа памяти

Любой переменной, объявленной с помощью директив описания простых типов данных, ассемблер присваивает три атрибута:

1. Сегмент (`seg`) – адрес начала сегмента, содержащего переменную.
2. Смещение (`offset`) в байтах от начала сегмента с переменной.
3. Тип (`type`) определяет количество памяти, выделяемой переменной в соответствии с директивой объявления переменной.

Получить и использовать значение этих атрибутов в программе можно с помощью рассмотренных нами выше операторов ассемблера `seg`, `offset` и `type`.

В заключение рекомендую вам обратить внимание на то, что в Справочнике приведены диапазоны значений для тех простых типов данных, директивы описания которых мы обсудили на данном уроке. Посмотрите эту информацию и используйте ее при необходимости явного задания значений для данных различных типов.

Подведем некоторые итоги:

- Программа на ассемблере, отражая особенности архитектуры микропроцессора, состоит из сегментов – блоков памяти, допускающих независимую адресацию.
- Каждый сегмент может состоять из предложений языка ассемблера четырех типов: команд ассемблера, макрокоманд, директив ассемблера и строк комментариев.
- Ассемблер допускает большое разнообразие типов операндов, которые могут содержаться непосредственно в команде, в регистрах и в памяти.
- Операнды в команде могут быть выражениями.
- Исходный текст программы разбивается на сегменты с помощью директив сегментации, которые делятся на стандартные, поддерживаемые всеми трансляторами ассемблера, и упрощенные, поддерживаемые транслятором TASM.
- Упрощенные директивы сегментации позволяют унифицировать интерфейс с языками высокого уровня и облегчают разработку программ, повышая наглядность кода.
- Существуют два режима работы TASM: MASM и IDEAL. Назначение режима MASM – обеспечить полную совместимость с транслятором MASM фирмы Microsoft. Назначение режима IDEAL – упростить синтаксис конструкций языка, повысить эффективность и скорость работы транслятора.
- TASM поддерживает разнообразные типы данных, которые делятся на простые (базовые) и сложные. Простые типы служат как бы «кирпичиками» для построения сложных типов данных.
- Директивы описания простых типов данных позволяют зарезервировать и при необходимости инициализировать области памяти заданной длины.
- Каждой переменной, объявленной с помощью директивы описания данных, TASM назначает атрибуты, доступ к которым можно получить с помощью соответствующих операторов ассемблера.

6

УРОК

Система команд микропроцессора

-
- Системы счисления
 - Алгоритм перевода чисел в различные системы счисления
 - Структура машинной команды
 - Способы адресации operandов
 - Общая характеристика системы команд
-

На этом уроке мы закончим обсуждение общих вопросов, необходимых для понимания программ на ассемблере. Будет рассмотрен формат машинной команды, мы научимся адресовать операнды в памяти и разберем основные способы адресации. Кроме того, мы покажем, как адресация операндов машинной команды отражается на ее структуре. В заключение будет рассмотрена классификация команд по их функциональному назначению. Это даст нам возможность оценить возможности микропроцессора по обработке данных.

Но прежде чем приступить к обсуждению, необходимо выяснить еще один принципиальный для программирования на ассемблере момент — работу с различными *системами счисления*. На уроке 3 при разборе программы мы затронули эту проблему. Актуальность ее при программировании на низком уровне очевидна из следующих положений:

- компьютер работает с двоичной информацией;
- человеку удобнее интерпретировать двоичную информацию посредством шестнадцатеричной системы счисления;
- человеку удобно производить вычисления, используя десятичную систему счисления.

Вам не раз в дальнейшем придется убедиться в истинности этих положений при работе с дампами памяти в отладчике или в других ситуациях. На практике при отладке или исследовании работы некоторой программы часто приходится заниматься интерпретацией содержимого нужного регистра или участка памяти, и поначалу проблема перевода чисел, как правило, вызывает определенные трудности. В связи с этим рассмотрим используемые при работе с компьютером системы счисления и алгоритмы взаимного преобразования чисел для этих систем счисления.

Системы счисления

Как известно, *системой счисления* называется совокупность правил записи чисел. Системы счисления подразделяются на *позиционные* и *непозиционные*. Как позиционные, так и непозиционные системы счисления используют определенный набор символов — *цифр*, последовательное сочетание которых образует число. Непозиционные системы счисления появились раньше позиционных. Они характеризуются тем, что в них символы, обозначающие то или иное число, не меняют своего значения в зависимости от местоположения в записи этого числа.

Классическим примером такой системы счисления является *римская*. В ней для записи чисел используются буквы латинского алфавита. При этом буква I означает единицу, V – пять, X – десять, L – пятьдесят, C – сто, D – пятьсот, M – тысячу. Для получения количественного эквивалента числа в римской системе счисления необходимо просто просуммировать количественные эквиваленты входящих в него цифр. Исключение из этого правила составляет случай, когда младшая цифра идет перед старшей, – в этом случае нужно не складывать, а вычитать число вхождений этой младшей цифры. К примеру, количественный эквивалент числа 577 в римской системе счисления – это $DLXXVII = 500 + 50 + 10 + 10 + 5 + 1 + 1 = 577$. Другой пример: $CDXXIX = 500 - 100 + 10 + 10 - 1 + 10 = 429$.

В позиционной системе счисления количество символов в наборе равно *основанию* системы счисления. Место каждой цифры в числе называется *позицией*. Номер позиции символа (за вычетом единицы) в числе называется *разрядом*. Разряд 0 называется *младшим* разрядом. Каждой цифре соответствует определенный количественный эквивалент. Введем обозначение – запись $A_{(p)}$, будет означать количественный эквивалент числа A, состоящего из n цифр a_k (где $k = 0, \dots, n-1$) в системе счисления с основанием p . Это число можно представить в виде последовательности цифр: $A_{(p)} = a_{n-1}a_{n-2}\dots a_1a_0$. При этом, конечно, всегда выполняется неравенство $a_k < p$.

В общем случае количественный эквивалент некоторого положительного числа A в позиционной системе счисления можно представить выражением:

$$A_{(p)} = a_{n-1} * p^{n-1} + a_{n-2} * p^{n-2} + \dots + a_1 * p^1 + a_0 * p^0, \quad (6.1)$$

где: p – основание системы счисления (некоторое целое положительное число); a – цифра данной системы счисления; n – номер старшего разряда числа.

Для получения количественного эквивалента числа в некоторой позиционной системе счисления необходимо сложить произведения количественных значений цифр на степени основания, показатели которых равны номерам разрядов (обратите внимание, что нумерация разрядов начинается с нуля).

После такого формального введения можно приступить к обсуждению некоторых позиционных систем счисления, наиболее часто используемых при разработке программ на ассемблере.

Двоичная система счисления

Набор цифр для двоичной системы счисления:

$\{0, 1\}$, основание степени $p = 2$.

Количественный эквивалент некоторого целого n -значного двоичного числа вычисляется согласно формуле (6.1):

$$A_{(2)} = a_{n-1} * 2^{n-1} + a_{n-2} * 2^{n-2} + \dots + a_1 * 2^1 + a_0 * 2^0. \quad (6.2)$$

Как мы уже отмечали, наличие этой системы счисления обусловлено тем, что компьютер построен на логических схемах, имеющих в своем элементарном виде только два состояния – включено и выключено. Производить счет в двоичной системе просто для компьютера, но сложно для человека. Например, рассмотрим двоичное число 10100111.

Вычислим количественный эквивалент этого двоичного числа. Согласно формуле (6.2), это будет величина, равная следующей сумме:

$$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0.$$

Посчитайте сами, сколько получится.

Сложение и вычитание двоичных чисел (рис. 6.1) выполняется так же, как и для других позиционных систем счисления, например десятичной. Точно так же выполняются заем и перенос единицы из (в) старший разряд. К примеру:

| | | | | | | | | | |
|---|---|---|---|---|---------|---|---|---|------|
| 1 | 1 | 1 | 1 | 1 | перенос | 1 | 1 | 1 | заем |
| + | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| + | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | | | | | | | |

Рис. 6.1. Сложение и вычитание двоичных чисел

Приведем степени двойки (табл. 6.1) и соответствие двоичных чисел и их десятичных и шестнадцатеричных эквивалентов (табл. 6.2).

Таблица 6.1. Степени двойки

| k | 2^k |
|----|-------|
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |
| 11 | 2048 |
| 12 | 4096 |

Шестнадцатеричная система счисления

Данная система счисления имеет следующий набор цифр:

{0, 1, 2, ..., 9, A, B, C, D, E, F}, основание системы $p = 16$.

Количественный эквивалент некоторого целого n -значного шестнадцатеричного числа вычисляется согласно формуле (6.1):

$$A_{(16)} = a_{n-1} \cdot 16^{n-1} + a_{n-2} \cdot 16^{n-2} + \dots + a_1 \cdot 16^1 + a_0 \cdot 16^0.$$

К примеру, количественный эквивалент шестнадцатеричного числа f45ed23c равен:

$$15 \cdot 16^7 + 4 \cdot 16^6 + 5 \cdot 16^5 + 14 \cdot 16^4 + 13 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 12 \cdot 16^0.$$

Посчитайте сами, сколько получится.

Таблица 6.2. Шестнадцатеричные цифры

| Десятичное число | Двоичная тетрада | Шестнадцатеричное число |
|------------------|------------------|-------------------------|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | A, a |
| 11 | 1011 | B, b |
| 12 | 1100 | C, c |
| 13 | 1101 | D, d |
| 14 | 1110 | E, e |
| 15 | 1111 | F, f |
| 16 | 10000 | 10 |

Запомнить поначалу эти соотношения сложно, поэтому полезно иметь под руками некоторую справочную информацию. Табл. 6.2 содержит представления десятичных чисел из диапазона 0–16 в двоичной и шестнадцатеричной системах счисления. Табл. 6.2 удобно использовать для взаимного преобразования чисел в рассматриваемых нами трех системах счисления. Шестнадцатеричная система счисления при производстве вычислений несколько сложнее, чем двоичная, в частности, в том, что касается правил переносов в старшие разряды (заемов из старших разрядов). Главное здесь — помнить следующее равенство:

$$(1 + F = 10)_{16}.$$

Эти переходы очень важны при выполнении сложения и вычитания шестнадцатеричных чисел. Пример приведен на рис. 6.2:

| | | | |
|-----------|-------------|-----------|-------------|
| 1 1 | перенос | 1 1 | заем |
| + E F 1 5 | 1 слагаемое | - B C D 8 | уменьшаемое |
| + C 1 E 8 | 2 слагаемое | - 5 E F 4 | вычитаемое |
| 1 B 0 F D | результат | 5 D E 4 | результат |

Рис. 6.2. Сложение и вычитание шестнадцатеричных чисел

Десятичная система счисления

Это наиболее известная система счисления, так как она постоянно используется нами в повседневной жизни. Данная система счисления имеет следующий набор цифр:

$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, основание степени $p = 10$.

Количественный эквивалент некоторого целого n -значного десятичного числа вычисляется согласно формуле (6.1):

$$A_{(10)} = a_{n-1} * 10^{n-1} + a_{n-2} * 10^{n-2} + \dots + a_1 * 10^1 + a_0 * 10^0.$$

К примеру, значение числа $A_{(10)} = 4523$ равно:

$$4 * 10^3 + 5 * 10^2 + 2 * 10^1 + 3 * 10^0.$$

Перевод чисел из одной системы счисления в другую

Рассмотрение данных систем счисления само по себе еще только полдела. Для того чтобы в полной мере использовать их в своей практической работе при программировании на ассемблере, необходимо научиться выполнять взаимное преобразование чисел между тремя системами счисления. Этим мы и займемся в дальнейшем. Кроме того, дополнительно будут рассмотрены некоторые особенности микропроцессоров Intel при работе с числами со знаком.

Перевод в десятичную систему счисления

Этот тип перевода наиболее прост. Обычно его производят с помощью так называемого *алгоритма замещения*, суть которого заключается в следующем: сначала в десятичную систему счисления переводится основание степени p , а затем — цифры исходного числа. Результаты подставляются в формулу (6.1). Полученная сумма и будет искомым результатом. Неявно при обсуждении двоичной и шестнадцатеричной систем счисления мы производили как раз такое преобразование.

Перевод в двоичную систему счисления

Перевод из десятичной системы счисления

1. Разделить десятичное число A на 2. Запомнить частное q и остаток a .
2. Если в результате шага 1 частное $q \neq 0$, то принять его за новое делимое и отметить остаток a , который будет очередной значащей цифрой числа, вернуться к шагу 1, на котором в качестве делимого (десятичного числа) участвует полученное на шаге 2 частное.

3. Если в результате шага 1 частное $q = 0$, алгоритм прекращается. Выписать остатки в порядке, обратном их получению. Получится двоичный эквивалент исходного числа.

К примеру, требуется перевести число 247_{10} в двоичную систему счисления (рис. 6.3):

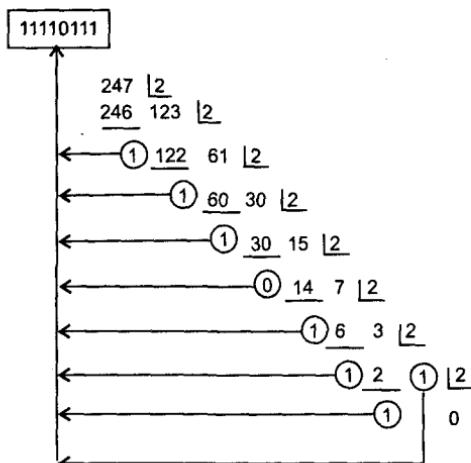


Рис. 6.3. Перевод в двоичную систему счисления

Порядок обхода остатков для получения результата показан стрелками, и результат преобразования равен 11110111_2 .

Перевод из шестнадцатеричной системы счисления

Этот переход мы уже обсуждали выше. Суть его заключается в последовательной замене шестнадцатеричных цифр соответствующими двоичными тетрадами согласно табл. 6.2. К примеру, $e4d5_{16} \rightarrow 1110\ 0100\ 1101\ 0101_2$.

Перевод в шестнадцатеричную систему счисления

Перевод из десятичной системы счисления

Общая идея такого перевода аналогична рассмотренной выше в алгоритме перевода в двоичную систему счисления из десятичной.

1. Разделить десятичное число A на 16. Запомнить частное q и остаток a .
2. Если в результате шага 1 частное $q \neq 0$, то принять его за новое делимое, записать остаток и вернуться к шагу 1.
3. Если частное $q = 0$, прекратить работу алгоритма. Выписать остатки в порядке, обратном их получению. Получится шестнадцатеричный эквивалент исходного десятичного числа.

К примеру, требуется преобразовать $32\ 767_{10}$ в шестнадцатеричную систему счисления (рис. 6.4).

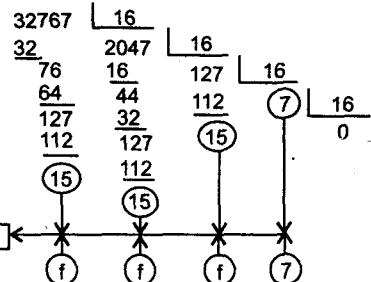


Рис. 6.4. Перевод в шестнадцатеричную систему счисления

Порядок обхода остатков для получения результата показан на рис. 6.4 стрелками. Результат преобразования равен 7fff_{16} .

Перевод из двоичной системы счисления

Идея алгоритма аналогична идее перевода из двоичной системы счисления в шестнадцатеричную. Суть в том, что двоичное число разбивается на тетрады, начиная с младшего разряда. Далее каждая тетрада приводится к соответствующему шестнадцатеричному числу согласно табл. 6.2.

К примеру, требуется перевести число

$11100101101011110101100011011000111101010101101_2$

в шестнадцатеричную систему счисления.

Разобьем его на тетрады:

$0111 \ 0010 \ 1101 \ 0111 \ 1010 \ 1100 \ 0110 \ 1100 \ 0111 \ 1010 \ 1010 \ 1101$.

По тетрадам приводим последовательности нулей и единиц к шестнадцатеричному представлению:

7 2 d 7 a c 6 c 7 a a d.

В итоге мы получили результат преобразования:

$11100101101011110101100011011000111101010101101_2 \rightarrow 72d7ac6c7aad_{16}$.

Перевод дробных чисел

Программист должен уметь выполнять перевод в различные системы счисления не только целых чисел, но и дробных. Особенно это важно при программировании алгоритмов, использующих операции с плавающей точкой. Без владения в полной мере знаниями о том, как представляются дробные числа в памяти компьютера и в регистрах сопроцессора, вам вряд ли удастся овладеть программированием на ассемблере в полной мере. Давайте разберемся с наиболее часто используемыми на практике способами перевода дробных чисел. Для этого формулу (6.1) преобразуем к следующему виду:

$$A_{(p)} = a_{n-1} * p^{n-1} + a_{n-2} * p^{n-2} + \dots + a_1 * p^1 + a_0 * p^0 a_{-1} * p^{-1} + a_{-2} * p^{-2} + \dots + a_{-m} * p^{-m}. \quad (6.3)$$

Рассмотрим операции перевода чисел на примерах.

Пример 6.1

Перевести в десятичное представление дробь в двоичной системе счисления $110100,01001011_2$.

Для перевода используем формулу (6.3):

$$110100,01001011_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} \\ + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8}.$$

Вычислить целую часть десятичной дроби $1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$ для вас труда не составит. Для вычисления остальной части выражения удобно использовать табл. 6.3.

Таблица 6.3. Значения отрицательных степеней по основанию числа 2

| m | 2^m |
|---|-------------------|
| 1 | 0,5 |
| 2 | 0,25 |
| 3 | 0,125 |
| 4 | 0,0625 |
| 5 | 0,03125 |
| 6 | 0,015625 |
| 7 | 0,0078125 и т. д. |

Далее вы сами можете продолжить табл. 6.3 значениями отрицательных степеней по основанию числа 2.

Подсчитайте результат перевода значения $110100,01001011_2$ в десятичное представление.

Пример 6.2

Перевести в десятичное представление дробь в шестнадцатеричной системе счисления $1df2,a1e4_{16}$.

Вновь используем формулу (6.3) и получим:

$$1df2,a1e4_{16} = 1 \cdot 16^3 + 13 \cdot 16^2 + 15 \cdot 16^1 + 2 \cdot 16^0 + 10 \cdot 16^{-1} + 1 \cdot 16^{-2} + 14 \cdot 16^{-3} + 4 \cdot 16^{-4}.$$

Для удобства вычисления дробной части приведем значения отрицательных степеней числа 16 табл. 6.4.

Таблица 6.4. Значения отрицательных степеней по основанию числа 16

| m | 16^m |
|---|-------------------------------|
| 1 | 0,0625 |
| 2 | 0,00390625 |
| 3 | 0,000244140625 |
| 4 | 0,0000152587890625 |
| 5 | 0,00000095367431640625 |
| 6 | 0,000000059604644775390625 |
| 7 | 0,000000037252902984619140625 |

Перевод из двоичной системы счисления в шестнадцатеричную систему и обратно выполняется, как обычно, на основе тетрад и трудности вызывать не должен. Рассмотрим теперь проблему представления десятичных дробей в двоичной и шестнадцатеричной системах счисления.

Общий алгоритм перевода десятичной дроби в другую систему счисления можно представить следующей последовательностью шагов:

1. Выделить целую часть десятичной дроби и выполнить ее перевод в выбранную систему счисления по алгоритмам, рассмотренным выше.
2. Выделить дробную часть и умножить ее на основание выбранной новой системы счисления.
3. В полученной после умножения дробной части десятичной дроби выделить целую часть и принять ее в качестве значения первого после запятой разряда числа в новой системе счисления.
4. Если дробная часть значения, полученного после умножения, равна нулю, то прекратить процесс перевода. Процесс перевода можно прекратить также в случае, если достигнута необходимая точность вычисления. В противном случае перейти к шагу 3.

Рассмотрим пример.

Пример 6.3

Перевести в двоичную систему счисления десятичную дробь 108.406_{10} .

1. Переведем целую часть десятичной дроби 108.406_{10} в двоичную систему счисления (рис. 6.5).

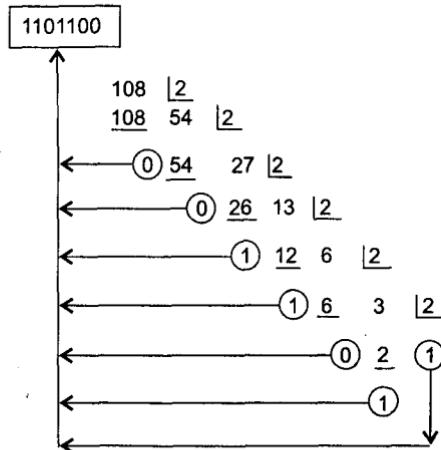


Рис. 6.5. Перевод целой части десятичного числа 108.406_{10} в двоичную систему счисления

2. Переведем дробную часть десятичного числа 108.406_{10} по алгоритму, приведенному выше (рис. 6.6).

011001111

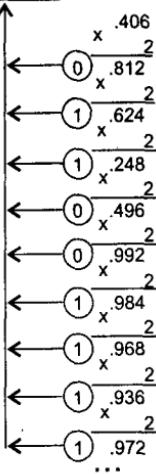


Рис. 6.6. Перевод дробной части числа 108.406_{10} в двоичную систему счисления

Результат перевода следующий: $108.406_{10} = 1101100.011001111_2$.

При переводе дробного числа из десятичной системы счисления в шестнадцатеричную целесообразно предварительно перевести число в двоичную систему. Затем двоичное представление разбить на тетрады отдельно до запятой и после запятой. Разбиение на тетрады двоичных разрядов целой части производят от запятой в сторону старших разрядов. Неполную старшую тетраду дополняют слева ведущими нулями. Разряды дробной части, напротив, разбивают на тетрады от запятой вправо к младшим разрядам. Если последняя тетрада неполная, то ее дополняют справа нулями. На рис. 6.7 показан процесс перевода дробного десятичного числа (пример 6.3) в эквивалентное шестнадцатеричное представление.

$$108,406 = 110 \boxed{1100,0110|0111}1 = 6c,678$$

| | | | | |
|---------|----------------|---|-------|---|
| (0) 110 | 1100,0110 0111 | 1 | (000) | |
| 6 | c | 6 | 7 | 8 |

Рис. 6.7. Пример перевода десятичного числа в шестнадцатеричную систему счисления

Числа со знаком

До сих пор предполагалось, что числа положительные. А как представляются в компьютере числа со знаком?

На уроке 2, обсуждая типы данных, мы отмечали, что целые числа могут представляться как числа со знаком и без знака. В чем отличие?

Положительные целые со знаком — это 0 и все положительные числа.

Отрицательные целые со знаком — это все числа, меньшие 0. Отличительным признаком числа со знаком является особая трактовка старшего бита поля, представляющего число. В качестве поля могут выступать байт, слово или двойное слово. Естественно, что физически этот бит ничем не отличается от других, — все зависит от команды, работающей с данным полем. Если в ее алгоритме заложена работа с целыми числами со знаком, то она будет по-особому трактовать старший бит поля. В случае если бит равен 0, число считается положительным и его значение вычисляется по правилам, которые мы рассмотрели выше. В случае если этот бит равен 1, число считается отрицательным и предполагается, что оно записано в так называемом *дополнительном коде*. Разберемся в том, что он собой представляет.

Дополнительный код некоторого отрицательного числа представляет собой результат инвертирования (замены 1 на 0 и наоборот) каждого бита двоичного числа, равного модулю исходного отрицательного числа плюс единица. К примеру, рассмотрим десятичное число -185_{10} . Модуль данного числа в двоичном представлении равен 10111001_2 . Сначала нужно дополнить это значение слева нулями до нужной размерности — байта, слова и т. д. В нашем случае дополнить нужно до слова, так как диапазон представления знаковых чисел в байте составляет $-128\dots127$. Следующее действие — получить *двоичное дополнение*. Для этого все разряды двоичного числа нужно инвертировать:

$$0000000010111001_2 \rightarrow 111111101000110_2$$

Теперь прибавляем единицу:

$$111111101000110_2 + 0000000000000001_2 = 111111101000111_2$$

Результат преобразования равен 111111101000111_2 . Именно так и представляется число -185_{10} в компьютере.

При работе с числами со знаком от вас наверняка потребуется умение выполнять обратное действие — имея двоичное дополнение числа, определить значение его модуля. Для этого необходимо выполнить два действия:

1. Выполнить инвертирование битов двоичного дополнения.

2. К полученному двоичному числу прибавить двоичную единицу.

К примеру, определим модуль двоичного представления числа $-185_{10} = 111111101000111_2$:

$$111111101000111_2 \rightarrow \text{инвертируем биты} \rightarrow 0000000010111000_2$$

Добавляем двоичную единицу:

$$0000000010111000_2 + 0000000000000001_2 = 0000000010111001_2 = |-185|$$

Теперь вам должно быть понятно, откуда появилась разница в диапазонах значений для чисел со знаком и без знака простых типов данных, обсуждавшихся на уроке 2.

Теперь мы почти готовы разговаривать с компьютером на его языке, состоящем из команд и данных. С данными мы уже разобрались, теперь давайте обратимся к командам.

Структура машинной команды

Машинная команда представляет собой закодированное по определенным правилам указание микропроцессору на выполнение некоторой операции или действия. Каждая команда содержит элементы, определяющие:

- *что делать?* (ответ на этот вопрос дает элемент команды, называемый *кодом операции* (КОП));
- *объекты, над которыми нужно что-то делать* (*операнды*);
- *как делать?* (эти элементы являются *типами operandов* и обычно задаются неявно).

Приведенный на рис. 6.8 формат машинной команды является самым общим. Максимальная длина машинной команды — 15 байт. Реальная команда может содержать гораздо меньшее количество полей, вплоть до одного — только КОП.

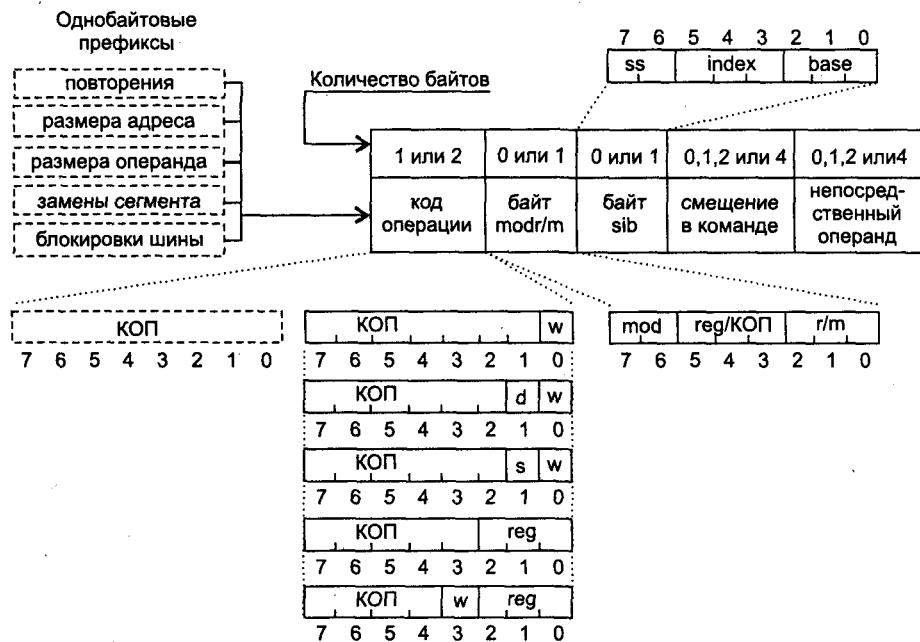


Рис. 6.8. Формат машинной команды

Опишем назначения полей машинной команды.

Префиксы. Необязательные элементы машинной команды, каждый из которых состоит из одного байта или может отсутствовать. В памяти префиксы предшествуют команде. Назначение префиксов — модифицировать операцию, выполняемую командой. Прикладная программа может использовать следующие типы префиксов.

Префикс замены сегмента. В явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию. Префиксы замены сегмента имеют следующие значения: 2eh — замена сегмента cs, 36h — замена сегмента ss, 3eh — замена сегмента ds, 26h — замена сегмента es, 64h — замена сегмента fs, 65h — замена сегмента gs.

Префикс разрядности адреса уточняет разрядность адреса (32- или 16-разрядный). Каждой команде, в которой используется адресный операнд, ставится в соответствие разрядность адреса этого операнда. Этот адрес может иметь разрядность 16 или 32 бит. Если разрядность адреса для данной команды 16 бит, это означает, что команда содержит 16-разрядное смещение (см. рис. 6.8) и оно соответствует 16-разрядному смещению адресного операнда относительно начала некоторого сегмента. В контексте рис. 2.4 и 2.5 это смещение называется *эффективным адресом*. Если разрядность адреса 32 бит, это означает, что команда содержит 32-разрядное смещение (см. рис. 6.8), оно соответствует 32-разрядному смещению адресного операнда относительно начала сегмента и по его значению формируется 32-битное смещение в сегменте. С помощью префикса разрядности адреса можно изменить действующее по умолчанию значение разрядности адреса. Это изменение будет касаться только той команды, которой предшествует префикс.

Префикс разрядности операнда аналогичен префиксу разрядности адреса, но указывает на разрядность операндов (32- или 16-разрядные), с которыми работает команда. В соответствии с какими правилами устанавливаются значения атрибутов разрядности адреса и операндов по умолчанию? В реальном режиме и режиме виртуального i8086 значения этих атрибутов — 16 бит. В защищенном режиме значения атрибутов зависят от состояния бита D в дескрипторе сегмента кода (см. урок 16). Если D = 0, то значения атрибутов, действующие по умолчанию, равны 16 бит; если D = 1, то 32 бита. Значения префиксов разрядности операнда 66h и разрядности адреса 67h. Вы можете с помощью префикса разрядности адреса в реальном режиме использовать 32-разрядную адресацию, но при этом необходимо помнить об ограниченности размера сегмента величиной 64 Кбайт. Аналогично префиксу разрядности адреса вы можете использовать префикс разрядности операнда в реальном режиме для работы с 32-разрядными операндами (к примеру, в арифметических командах).

Префикс повторения используется с цепочечными командами (командами обработки строк). Этот префикс «зацикливает» команду для обработки всех элементов цепочки. Система команд поддерживает два типа префиксов: *безусловные* (rep — 0f3h), заставляющие повторяться цепочечную команду некоторое количество раз, и *условные* (repe/repz — 0f3h, repne/repnz — 0f2h), которые при зацикливании проверяют некоторые флаги, — и в результате проверки возможен досрочный выход из цикла.

Код операции. Обязательный элемент, описывающий операцию, выполняемую командой. Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции.

Последующие поля машинной команды определяют местоположение операндов, участвующих в операции, и особенности их использования. Рассмотрение этих

полей связано со способами задания операндов в машинной команде и потому будет выполнено ниже.

Байт режима адресации modr/m. Значение этого байта определяет используемую форму адреса операндов. Операнды могут находиться в памяти, в одном или двух регистрах. Если операнд находится в памяти, то байт modr/m определяет компоненты (смещение, базовый и индексный регистры), используемые для вычисления его эффективного адреса (см. рис. 2.3). В защищенным режиме для определения местоположения операнда в памяти может дополнительно использоваться байт sib (Scale-Index-Base — масштаб-индекс-база). Байт modr/m состоит из трех полей (см. рис. 6.8):

- поле mod определяет количество байт, занимаемых в команде адресом операнда (см. рис. 6.8, поле смещение в команде). Поле mod используется совместно с полем r/m, которое указывает способ модификации адреса операнда смещение в команде. К примеру, если mod = 00, это означает, что поле смещение в команде отсутствует, и адрес операнда определяется содержимым базового и(или) индексного регистра. Какие именно регистры будут использоваться для вычисления эффективного адреса, определяется значением этого байта. Если mod = 01, это означает, что поле смещение в команде присутствует, занимает один байт и модифицируется содержимым базового и (или) индексного регистра. Если mod = 10, это означает, что поле смещение в команде присутствует, занимает два или четыре байта (в зависимости от действующего по умолчанию или определяемого префиксом размера адреса) и модифицируется содержимым базового и(или) индексного регистра. Если mod = 11, это означает, что операндов в памяти нет; они находятся в регистрах. Это же значение байта mod используется в случае, когда в команде применяется непосредственный операнд;
- поле reg/коп определяет либо регистр, находящийся в команде на месте второго операнда, либо возможное расширение кода операции;
- поле r/m используется совместно с полем mod и определяет либо регистр, находящийся в команде на месте первого операнда (если mod = 11), либо используемые для вычисления эффективного адреса (совместно с полем смещение в команде) базовые и индексные регистры.

Байт масштаб-индекс-база (байт sib) используется для расширения возможностей адресации операндов. На наличие байта sib в машинной команде указывает сочетание одного из значений 01 или 10 поля mod и значения поля r/m = 100. Байт sib состоит из трех полей:

- поле масштаба ss. В этом поле размещается масштабный множитель для индексного компонента index, занимающего следующие три бита байта sib. В поле ss может содержаться одно из следующих значений: 1, 2, 4, 8. При вычислении эффективного адреса на это значение будет умножаться содержимое индексного регистра. Более подробно, с практической точки зрения, эта расширенная возможность индексации рассматривается на уроке 12 при обсуждении вопросов работы с массивами;
- поле index используется для хранения номера индексного регистра, который применяется для вычисления эффективного адреса операнда;

- поле **base** используется для хранения номера базового регистра, который также применяется для вычисления эффективного адреса операнда. Напомню, что в качестве базового и индексного регистров могут использоваться практически все регистры общего назначения.

Поле смещения в команде. 8-, 16- или 32-разрядное целое число со знаком, представляющее собой, полностью или частично (с учетом вышеприведенных рассуждений), значение эффективного адреса операнда.

Поле непосредственного операнда. Необязательное поле, представляющее собой 8-, 16- или 32-разрядный непосредственный operand. Наличие этого поля, конечно, отражается на значении байта **mopr/m**.

Способы задания operandов команды

В ходе предыдущего изложения мы поневоле касались вопроса о том, где располагаются operandы, с которыми работает машинная команда, и как это отражается на содержимом ее полей. В этой части урока мы рассмотрим этот вопрос более систематизированно и в полном объеме. Это позволит нам уже со следующего урока перейти непосредственно к практическим вопросам программирования на языке ассемблера.

Операнд задается неявно на микропрограммном уровне. В этом случае команда явно не содержит operandов. Алгоритм выполнения команды использует некоторые объекты по умолчанию (registры, флаги в **eflags** и т. д.). Например, команды **cli** и **sti** неявно работают с флагом прерывания **if** в регистре **eflags**, а команда **xlat** неявно обращается к регистру **al** и строке в памяти по адресу, определяемому парой регистров **ds:bx**.

Операнд задается в самой команде (непосредственный операнд). Операнд находится в коде команды, то есть является ее частью. Для хранения такого операнда в команде выделяется поле длиной до 32 бит (см. рис. 6.8). Непосредственный операнд может быть только вторым operandом (источником). Операнд-получатель может находиться либо в памяти, либо в регистре. Например, **mov ax,0ffffh** пересыпает в регистр **ax** шестнадцатеричную константу **ffff**. Команда **add sum,2** складывает содержимое поля по адресу **sum** с целым числом 2 и записывает результат по месту первого операнда, то есть в память.

Операнд находится в одном из регистров. Регистровые operandы указываются именами регистров. В качестве регистров могут использоваться:

- 32-разрядные регистры **EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP;**
- 16-разрядные регистры **AX, BX, CX, DX, SI, DI, SP, BP;**
- 8-разрядные регистры **AH, AL, BH, BL, CH, CL, DH, DL;**
- сегментные регистры **CS, DS, SS, ES, FS, GS.**

Например, команда **add ax,bx** складывает содержимое регистров **ax** и **bx** и записывает результат в **bx**. Команда **dec si** уменьшает содержимое **si** на 1.

Операнд располагается в памяти. Это наиболее сложный и в то же время наиболее гибкий способ задания operandов. Он позволяет реализовать следующие два основных вида адресации: прямую адресацию и косвенную адресацию.

В свою очередь, косвенная адресация имеет следующие разновидности:

- косвенная базовая адресация; другое ее название — регистровая косвенная адресация;
- косвенная базовая адресация со смещением;
- косвенная индексная адресация со смещением;
- косвенная базовая индексная адресация;
- косвенная базовая индексная адресация со смещением.

Операндом является порт ввода-вывода. Как мы уже отмечали, помимо адресного пространства оперативной памяти микропроцессор поддерживает адресное пространство ввода-вывода, которое используется для доступа к устройствам ввода-вывода. Объем адресного пространства ввода-вывода составляет 64 Кбайт. Для любого устройства компьютера в этом пространстве выделяются адреса. Конкретное значение адреса в пределах этого пространства называется *портом ввода-вывода*. Физически порту ввода-вывода соответствует аппаратный регистр (не путать с регистром микропроцессора), доступ к которому осуществляется с помощью специальных команд ассемблера *in* и *out*. Например:

```
in    a1,60h :ввести байт из порта 60h
```

Регистры, адресуемые с помощью порта ввода-вывода, могут иметь разрядность 8, 16 или 32 бит, но для конкретного порта разрядность регистра фиксированна. Команды *in* и *out* работают с фиксированной номенклатурой объектов. В качестве источника информации или получателя применяются так называемые *регистры-аккумуляторы* *eah*, *ax*, *a1*. Выбор регистра определяется разрядностью порта. Номер порта может задаваться непосредственным операндом в командах *in* и *out* или значением в регистре *dx*. Последний способ позволяет динамически определить номер порта в программе. Например:

```
mov  dx,20h ;записать номер порта 20h в регистр dx  
mov  a1,20h ;записать значение 20h в регистр a1  
out  dx,a1 :вывести значение 20h в порт 20H
```

Операнд находится в стеке.

Команды могут совсем не иметь операндов, иметь один или два операнда. Большинство команд требует двух операндов, один из которых является *операндом-источником*, а второй — *операндом назначения*. Важно то, что один операнд может располагаться в регистре или памяти, а второй операнд *обязательно* должен находиться в регистре или непосредственно в команде. Непосредственный операнд может быть только операндом-источником.

В двухоперандной машинной команде возможны следующие сочетания операндов:

- регистр—регистр;
- регистр—память;
- память—регистр;
- непосредственный операнд—регистр;
- непосредственный операнд—память.

Для данного правила есть исключения, которые касаются:

- *команды работы с цепочками*, которые могут перемещать данные из памяти в память;
- *команды работы со стеком*, которые могут переносить данные из памяти в стек, также находящийся в памяти;
- *команды типа умножения*, которые, кроме операнда, указанного в команде, используют еще и второй, неявный операнд.

Из перечисленных сочетаний операндов наиболее часто употребляются *регистр—память* и *память—регистр*. Ввиду их важности рассмотрим их подробнее. Обсуждение мы будем сопровождать примерами команд ассемблера, которые будут показывать, как изменяется формат команды ассемблера при применении того или иного вида адресации. В связи с этим посмотрите еще раз на рис. 2.5, на котором показан принцип формирования физического адреса на адреснойшине микропроцессора. Видно, что адрес операнда формируется как сумма двух составляющих — сдвинутого на 4 бита содержимого сегментного регистра и 16-битного эффективного адреса, который в общем случае вычисляется как сумма трех компонентов: базы, смещения и индекса.

Рассмотрим особенности основных видов адресации операндов в памяти.

Прямая адресация

Это простейший вид адресации операнда в памяти, так как эффективный адрес содержится в самой команде и для его формирования не используется никаких дополнительных источников или регистров. Эффективный адрес берется непосредственно из поля смещения машинной команды (см. рис. 6.8), которое может иметь размер 8, 16, 32 бита. Это значение однозначно определяет байт, слово или двойное слово, расположенные в сегменте данных.

Прямая адресация может быть двух типов:

- *Относительная прямая адресация*. Используется для команд условных переходов, для указания относительного адреса перехода. Относительность такого перехода заключается в том, что в поле смещения машинной команды содержится 8-, 16- или 32-битное значение, которое в результате работы команды будет складываться с содержимым регистра указателя команд *ip/eip*. В результате такого сложения получается адрес, по которому и осуществляется переход.

К примеру:

```
jc m1    :переход на метку m1, если флаг cf = 1  
        mov    a1,2
```

...

m1:

Несмотря на то что в команде указана некоторая метка в программе, ассемблер вычисляет смещение этой метки относительно следующей команды (в нашем случае это *mov a1,2*) и подставляет его в формируемую машинную команду *jc*.

- **Абсолютная прямая адресация.** В этом случае эффективный адрес является частью машинной команды, но формируется этот адрес только из значения поля смещения в команде. Для формирования физического адреса операнда в памяти микропроцессор складывает это поле со сдвинутым на 4 бита значением сегментного регистра. В команде ассемблера можно использовать несколько форм такой адресации. К примеру:

```
mov ax,dword ptr [0000]      ;записать слово по адресу  
                           ;:ds:0000 в регистр ах
```

Но такая адресация применяется редко; обычно используемым ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в формируемую им машинную команду в поле смещение в команде (см. рис. 6.8). В итоге получается так, что машинная команда прямо адресует свой операнд, имея, фактически, в одном из своих полей значение эффективного адреса. К примеру:

```
data    segment  
per1    dw    5  
...  
data    ends  
code    segment  
    mov ax,data  
    mov ds,ax  
...  
    mov ax,per1      ;записать слово per1 (его физический адрес ds:0000) в ах
```

Мы получим тот же результат, что и при использовании команды

```
mov ax,dword ptr [0000]
```

Остальные виды адресации относятся к косвенным. Слово «косвенный» в названии этих видов адресации означает то, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах, на которые указывают своим содержимым байт modr/m и, возможно, байт sib.

Косвенная базовая (регистровая) адресация

При такой адресации эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме sp/esp и bp/ebp (это специфические регистры для работы с сегментом стека).

Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки []. К примеру, команда `mov ax,[есх]` помещает в регистр ах содержимое слова по адресу из сегмента данных со смещением, хранящимся в регистре есх.

Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это свойство очень полезно, например, для организации циклических вычислений и для работы с различными структурами данных типа таблиц или массивов.

Косвенная базовая (регистровая) адресация со смещением

Этот вид адресации является дополнением предыдущего и предназначен для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресаций удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее, на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться динамически, на стадии выполнения программы. Модификация содержимого базового регистра позволяет обратиться к одноименным элементам различных экземпляров однотипных структур данных.

К примеру, команда `mov ax,[edx+3h]` пересыпает в регистр `ax` слова из области памяти по адресу: содержимое `edx + 3h`. Команда `mov ax,mas[dx]` пересыпает в регистр `ax` слово по адресу: содержимое `dx` плюс значение идентификатора `mas` (не забывайте, что транслятор присваивает каждому идентификатору значение, равное смещению этого идентификатора относительно начала сегмента данных).

Косвенная индексная адресация со смещением

Этот вид адресации очень похож на косвенную базовую адресацию со смещением. Здесь также для формирования эффективного адреса используется один из регистров общего назначения. Но индексная адресация обладает одной интересной особенностью, которая очень удобна для работы с массивами. Она связана с возможностью так называемого масштабирования содержимого индексного регистра. Что это такое? Посмотрите на рис. 6.8. Нас интересует байт `sib`. При обсуждении структуры этого байта мы отмечали, что он состоит из трех полей. Одно из этих полей — поле `масштаба ss`, на значение которого умножается содержимое индексного регистра. К примеру, в команде `mov ax,mas[esi*2]` значение эффективного адреса второго операнда вычисляется выражением `mas+(esi)*2`. В связи с тем что в ассемблере нет средств для организации индексации массивов, программисту своими силами приходится ее организовывать. Наличие возможности масштабирования существенно помогает в решении этой проблемы, но при условии, что размер элементов массива составляет 1, 2, 4 или 8 байт.

Косвенная базовая индексная адресация

При этом виде адресации эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто используется масштабирование содержимого индексного регистра. Например:

`mov eax,[esi][edx]`

В данном примере эффективный адрес второго операнда формируется из двух компонентов, `(esi)+(edx)`.

Косвенная базовая индексная адресация со смещением

Этот вид адресации является дополнением косвенной индексной адресации. Эффективный адрес формируется как сумма трех составляющих: содержимого

базового регистра, содержимого индексного регистра и значения поля смещения в команде. К примеру, команда `mov eax,[esi+5][edx]` пересыпает в регистр eax двойное слово по адресу: (`esi`) + 5 + (`edx`). Команда `add ax,array[esi][ebx]` производит сложение содержимого регистра ax с содержимым слова по адресу: значение идентификатора array + (`esi`) + (`ebx`).

Функциональная классификация машинных команд

Система команд микропроцессора содержит более 300 машинных команд. С появлением каждой новой модели микропроцессора их количество, как правило, возрастает, отражая тем самым архитектурные нововведения, отличающие эту модель от ее предшественниц. Набор машинных команд можно структурировать по группам и подгруппам. Очень полезно перед началом изучения работы отдельных команд получить общее представление о всей системе команд микропроцессора (рис. 6.9).

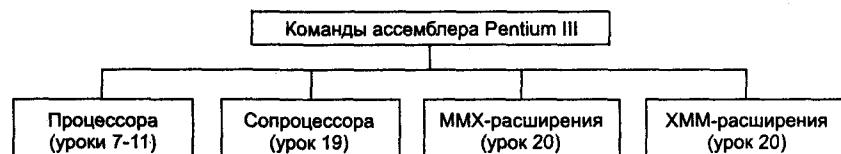


Рис. 6.9. Машинные команды микропроцессора Intel (до Pentium III)

На первых уроках мы будем рассматривать команды основного процессора (целочисленные), поэтому приведем их классификацию (рис. 6.10).

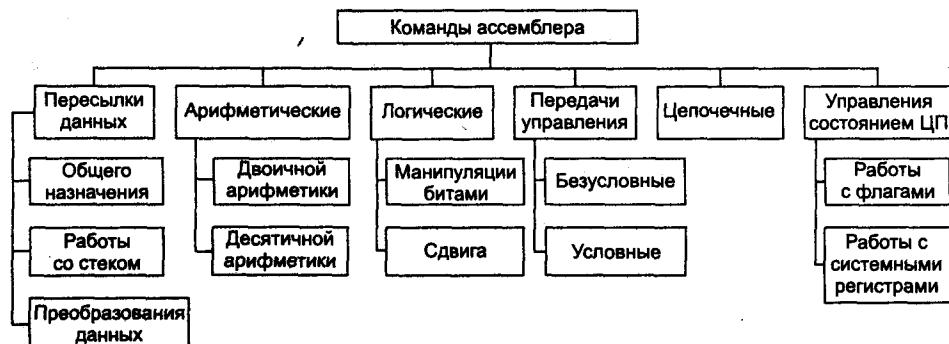


Рис. 6.10. Функциональная классификация целочисленных машинных команд

Детальную информацию обо всех командах микропроцессора можно найти в Справочнике. Его удобно использовать в ходе работы над программными

проектами. В нем вы найдете информацию о назначении, кодировке, флагах, алгоритмах выполнения команд и примеры их использования.

На этом урок 6 можно закончить. Начиная со следующего урока 7 мы приступим к практическим вопросам программирования на ассемблере.

Подведем некоторые итоги:

- Специфика работы программиста предполагает хорошее владение счетом в трех системах счисления: двоичной, шестнадцатеричной и десятичной. При этом программист должен достаточно бегло производить перевод чисел из одной системы счисления в другую.
- Числа со знаком представляются в компьютере особым образом: положительные числа — в виде обычного двоичного числа, а отрицательные — в дополнительном коде.
- Структура команд микропроцессора позволяет обеспечить большую гибкость при обработке операндов и разнообразие режимов адресации.



УРОК

Команды обмена данными

-
- Линейные алгоритмы
 - Команды пересылки данных
 - Ввод-вывод в порт
 - Команды работы с адресами памяти
 - Команды работы со стеком
-

Наверняка вы уже знакомы с понятием алгоритма, представляющего собой формализованное описание логики работы программы. Способы такой формализации весьма разнятся: от текстового описания последовательности действий до алгоритма развитых case-систем. Последовательность действий, описываемых алгоритмом, может быть:

- **линейной** — все действия выполняются поочередно, друг за другом;
- **нелинейной** — в алгоритме есть точки ветвления, в которых должно приниматься решение о месте, с которого программа продолжит свое выполнение. Решение может носить условный или безусловный характер.

Линейные участки алгоритма обычно содержат команды манипуляции данными, вычисления значений выражений, преобразования данных. В точках ветвления размещают команды сравнения, различных видов перехода, вызова подпрограмм и некоторые другие.

Обратимся к схеме на рис. 6.10, приведенной в конце урока 6. На ней показаны целочисленные команды микропроцессора. Из всей их совокупности на линейных участках работают следующие группы:

- команды пересылки данных;
- арифметические команды;
- логические команды;
- команды управления состоянием микропроцессора.

На этом уроке мы рассмотрим только группу команд пересылки данных. Эти команды осуществляют пересылку данных из одного места в другое, запись и чтение информации из портов ввода-вывода, преобразование информации, манипуляции с адресами и указателями, обращение к стеку. Для некоторых из этих команд операция пересылки является только частью алгоритма. Другая его часть выполняет некоторые дополнительные операции над пересылаемой информацией. Поэтому для удобства практического применения и отражения их специфики данные команды будут рассмотрены в соответствии с их функциональным назначением, согласно которому они делятся на команды:

- пересылки данных;
- ввода-вывода в порт;
- работы с адресами и указателями;
- преобразования данных;
- работы со стеком.

Пересылка данных

К этой группе относятся следующие команды:

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

`mov` — это основная команда пересылки данных. Она реализует самые разнообразные варианты пересылки. Отметим особенности применения этой команды:

- командой `mov` нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения. К примеру, рассмотрим фрагмент программы для пересылки байта из ячейки `f1s` в ячейку `f1d`:

```
masm
model small
.data
f1s db 5
fld db ?
.code
start:
...
    mov al,f1s
    mov fld,al
...
end start
```

- нельзя загрузить в сегментный регистр значение непосредственно из памяти. Поэтому для выполнения такой загрузки нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек. Если вы посмотрите листинги 3.1 и 5.1, то увидите в начале сегмента кода две команды `mov`, выполняющие настройку сегментного регистра `ds`. При этом из-за невозможности загрузить в прямую в сегментный регистр значение адреса сегмента, содержащееся в предопределенной переменной `@data`, приходится использовать регистр общего назначения `ax`;

- нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных все те же регистры общего назначения. Вот пример инициализации регистра `es` значением из регистра `ds`:

```
mov ax,ds
mov es,ax
```

Но есть и другой, более красивый способ выполнения данной операции — использование стека и команд `push` и `pop`:

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

- нельзя использовать сегментный регистр cs в качестве операнда назначения. Причина здесь простая. Дело в том, что в архитектуре микропроцессора пара cs:ip всегда содержит адрес команды, которая должна выполняться следующей. Изменение командой mov содержимого регистра cs фактически означало бы операцию перехода, а не пересылки, что недопустимо.

В связи с командой mov отметим один тонкий момент. Пусть в регистре bx содержится адрес некоторого поля (то есть мы используем косвенную базовую адресацию). Его содержимое нужно переслать в регистр ax. Очевидно, что нужно применить команду mov в форме:

```
mov ax,[bx]
```

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

Или другой случай, возможно, более показательный. Команды *инкремента inc* (увеличения операнда на 1) и *декремента dec* (уменьшения операнда на 1):

```
inc [bx]
```

...

```
dec [bx]
```

Что адресуется регистром bx в памяти — байт, слово или двойное слово?

Допустим также, что требуется инициализировать поле, адресуемое bx, значением 0. Очевидно, что одно из решений — применение mov:

```
mov [bx],0
```

И опять у транслятора вопрос: какую машинную команду ему конструировать — для инициализации байта, слова или двойного слова?

Во всех этих случаях необходимо уточнять тип используемых operandов. Для этого существует специальный оператор ассемблера ptr (см. урок 5). Правильно записать вышеупомянутые команды можно следующим образом:

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

Можно рекомендовать использовать оператор ptr во всех сомнительных относительно согласования размеров operandов случаях. Также этот оператор нужно применять, когда требуется принудительно поменять размерность operandов. К примеру, требуется переслать значение 0ffh во второй байт поля flp:

```
masm
model small
.data
...
flpdw 0
...
.code
start:
```

```
...    mov  byte ptr (flp+1),0ffh
```

```
...end start
```

Несмотря на то что поле `flp` имеет тип `word`, мы сообщаем ассемблеру, что поле нужно трактовать как байтовое, и заставляем вычислить значение эффективного адреса второго операнда как смещение `flp` плюс единица. Тем самым мы получаем доступ ко второму байту поля `flp`.

Для двунаправленной пересылки данных применяют команду `xchg`. Для этой операции можно, конечно, применить последовательность из нескольких команд `mov`, но из-за того, что операция обмена используется довольно часто, разработчики системы команд микропроцессора посчитали нужным ввести отдельную команду обмена `xchg`. Естественно, что операнды должны иметь один тип. Не допускается (как и для всех команд ассемблера) обменивать между собой содержимое двух ячеек памяти. К примеру:

```
xchg ax,bx          ;обменять содержимое регистров ax и bx  
xchg ax,word ptr [si] ;обменять содержимое регистра ax  
                      ;и слова в памяти по адресу в [si]
```

Ввод-вывод в порт

На уроке 6 при обсуждении вопроса о том, где могут находиться операнды машинной команды, мы упоминали *порт ввода-вывода*. Напомню основные моменты. Каждое устройство ввода-вывода, каждое системное устройство имеют один или несколько регистров, доступ к которым осуществляется через *адресное пространство ввода-вывода*. Эти регистры имеют разрядность 8-, 16- или 32 бит. Адресное пространство ввода-вывода физически независимо от пространства оперативной памяти и имеет ограниченный объем, составляющий 2^{16} , или 65 536 адресов ввода-вывода. Таким образом, понятие порта ввода-вывода можно определить как 8-, 16- или 32-разрядный аппаратный регистр, имеющий определенный адрес в адресном пространстве ввода-вывода. Вся работа системы с устройствами на самом низком уровне выполняется с использованием портов ввода-вывода. Посмотрите на рис. 7.1. На нем показана сильно упрощенная, концептуальная схема управления оборудованием компьютера.

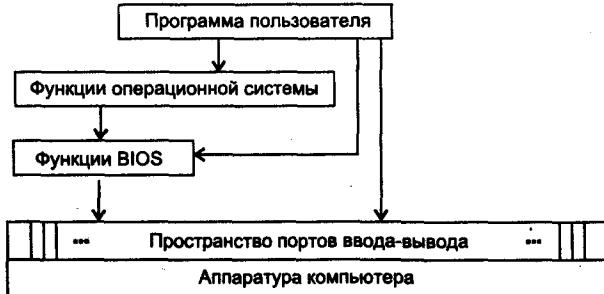


Рис. 7.1. Концептуальная схема управления оборудованием компьютера

Как видно из рис. 7.1, самым нижним уровнем является уровень BIOS, на котором работа с оборудованием ведется напрямую через порты. Тем самым реализуется концепция независимости от оборудования. При замене оборудования необходимо будет лишь подправить соответствующие функции BIOS, переориентировав их на новые адреса и логику работы портов.

Принципиально управлять устройствами напрямую через порты несложно. Сведения о номерах портов, их разрядности, формате управляющей информации приводятся в техническом описании на устройство. Необходимо знать лишь конечную цель своих действий, алгоритм, в соответствии с которым работает конкретное устройство, и порядок программирования его портов. То есть, фактически, нужно знать, что и в какой последовательности нужно послать в порт (при записи в него) или считать из него (при чтении) и как следует трактовать эту информацию. Для этого достаточно всего двух команд, присутствующих в системе команд микропроцессора:

in аккумулятор, номер_порта — ввод в аккумулятор из порта с номером *номер_порта*,
out порт, аккумулятор — вывод содержимого аккумулятора в порт с номером *номер_порта*.

Возможные значения операндов этих команд приведены в Справочнике.

В качестве примера рассмотрим, как на уровне аппаратуры заставить компьютер издавать звук сирены. Вначале мы перечислим, какие аппаратные ресурсы при этом будут задействованы и как ими надо управлять.

Вам, наверное, известно, что у компьютера есть внутренний динамик. Как это ни удивительно, но специальной схемы генерации звука для него нет. Сигнал для управления динамиком формируется в результате совместной работы микросхем:

- *программируемого периферийного интерфейса* (ППИ) i8255;
- таймера i8253.

Общая схема формирования такого сигнала показана на рис. 7.2.

Обсудим схему на рис. 7.2. Основная работа по генерации звука производится микросхемой таймера. Микросхема таймера (далее просто таймер) имеет три канала с совершенно одинаковыми внутренней структурой и принципом работы. На каналы таймера подаются импульсы от микросхемы системных часов, которые, по сути, представляют собой генератор импульсов, работающий с частотой 1,19 МГц. Каждый канал имеет два входа и один выход. Выходы канала замкнуты на вполне определенные устройства компьютера. Так, канал 0 замкнут на контроллер прерываний и является источником аппаратного прерывания от таймера, которое возникает 18,2 раза в секунду. Канал 1 связан с микросхемой прямого доступа к памяти (DMA). И наконец, канал 2 выходит на динамик компьютера. Как мы отметили, каналы таймера имеют одинаковую структуру, основу которой составляют три регистра: *регистр ввода-вывода* разрядностью 8 бит, *регистр-зашелка* (*latch register*) и *регистр-счетчик*, оба по 16 бит. Все регистры связаны между собой следующим образом. В регистр ввода-вывода извне помещается некоторое значение. Источником этого значения может быть либо системное программное обеспечение, либо программа пользователя. Каждый регистр ввода-вывода имеет адрес в адресном пространстве ввода-вывода (номер порта вво-

да-вывода). Регистр ввода-вывода канала 2 имеет номер порта ввода-вывода 42h. Помещаемые в него значения немедленно попадают в регистр-защелку, или, как его еще называют, *регистр-фиксатор*, где значение сохраняется до тех пор, пока в регистр ввода-вывода не будет записано новое значение. Но как, спросите вы, согласуются эти регистры по их разрядности, ведь один из них 8-, а другой 16-разрядный? Для этого предназначен *регистр управления* (ему соответствует порт 43h), который является частью механизма управления всей микросхемой таймера. Он содержит слово состояния, с помощью которого производятся выбор каналов, задание режима работы канала и тип операции передачи значения в канал.

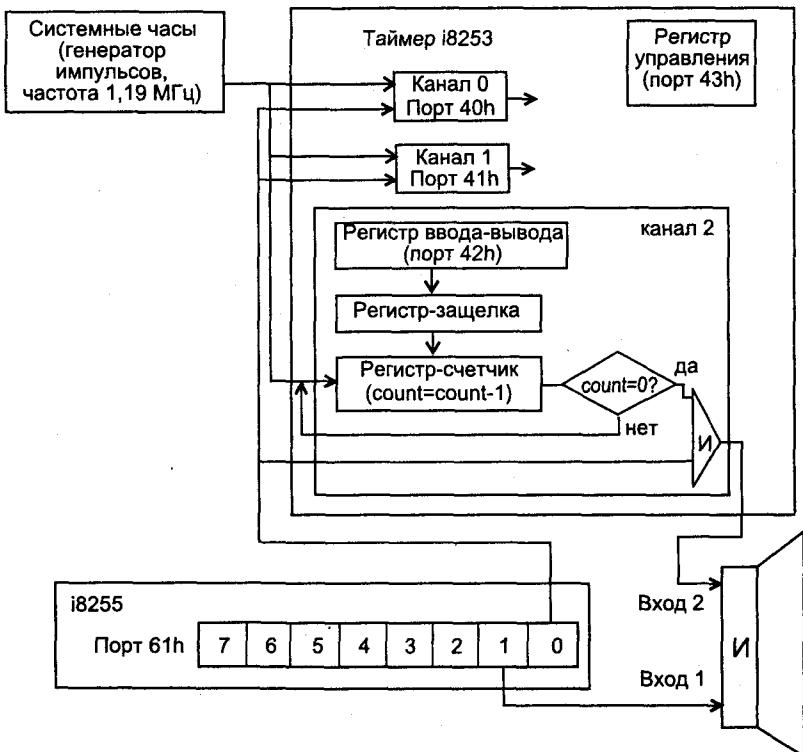


Рис. 7.2. Схема формирования звука для встроенного динамика

Слово состояния имеет следующую структуру:

- бит 0 определяет тип константы пересчета: 0 — константа задана двоичным числом, 1 — константа задана двоично-десятичным (BCD) числом. Константа пересчета — значение, загружаемое извне в регистр-защелку; в нашем случае загружаться будет двоичное число, поэтому значение этого поля будет 0;
- биты 1–3 определяют режим работы микросхемы таймера. Всего можно определить шесть режимов, но обычно используется третий, поэтому для нашего случая значение поля — 011;

О биты 4–5 определяют тип операции: 00 — передать значение счетчика в регистр-задвижку (то есть возможна не только операция записи значения в канал, но и извлечения значения регистра-счетчика из него), 10 — записать в регистр-задвижку только старший байт, 01 — записать в регистр-задвижку только младший байт, 11 — записать в регистр-задвижку сначала старший байт, затем младший. В нашем случае значение поля будет 11. Поэтому формирование 16-битного регистра-защелки через 8-битный регистр ввода-вывода производится следующим образом: запись производится в два приема, первый байт из регистра ввода-вывода записывается на место старшего байта регистра-защелки, второй байт — на место младшего байта. Нетрудно догадаться, что в регистр ввода-вывода эти байты помещаются командами `in` и `out`;

О биты 6–7 определяют номер программируемого канала. В нашем случае они равны 10.

Для формирования любого звука необходимо задать его длительность и высоту. После того как значение из регистра ввода-вывода попало в регистр-защелку, оно моментально записывается в регистр-счетчик. Сразу же после этого значение регистра-счетчика начинает уменьшаться на единицу с приходом каждого импульса от системных часов. На выходе любого из трех каналов таймера стоит схема логического умножения. Эта схема имеет два входа и один выход. Значение регистра-счетчика участвует в формировании сигнала на одном из входов схемы логического умножения И. Сигнал на втором входе этой схемы зависит от состояния бита 0 регистра микросхемы интерфейса с периферией (порт 61h). В свое время мы подробно разберемся с логическими операциями, сейчас следует лишь пояснить, что единица на выходе схемы логического умножения может появиться только в одном случае — когда на обоих входах единицы. Когда значение в регистре-счетчике становится равным нулю, на соответствующем входе схемы И формируется такая единица. И если при этом на втором входе, значение которого зависит от бита 0 порта 61h, также 1, то импульс от системных часов проходит на выход канала 2. Одновременно с пропуском импульса в канале 2 немедленно производится загрузка содержимого регистра-защелки (которое не изменилось, если его не изменили извне) в регистр-счетчик. Весь процесс с уменьшением содержимого регистра-счетчика повторяется заново. Теперь вы понимаете, что чем меньшее значение загружено в регистр-защелку, тем чаще будет обнуление регистра-счетчика, и тем чаще импульсы будут проходить на выход канала 2. А это означает, что *высота* звука будет выше. Понятно, что максимальное значение частоты на входе 1 динамика — 1,19 МГц. Таким образом импульс с выхода канала 2 попадает на динамик, и если на последний подан ток, то возникает долгожданный звук. Подачей тока на динамик управляет бит 1 порта 61h. Как прервать звучание? Очевидно, что для этого возможны два пути: первый — отключить ток, сбросив бит 1 порта 61h, второй — сбросить бит 0 порта 61h. Эти две возможности используют для создания различных звуковых эффектов. Выполняя эти разрывы, мы фактически определяем *длительность* звучания.

Если вы внимательно следили за всеми рассуждениями, то, наверное, без труда сможете понять, почему первый канал таймера формирует сигналы аппаратного

прерывания от таймера 18,2 раза в секунду (на основании этих сигналов программы отслеживают время). Для этого BIOS во время загрузки после включения компьютера загружает в первый канал соответствующее значение.

Таким образом, наметились три последовательных действия, необходимые для программирования звукового канала таймера (они применимы и к остальным каналам):

1. Посредством порта 43h выбрать канал, задать режим работы и тип операции передачи значения в канал. В нашем случае соответствующее значение будет равно 10110110 = 0b6h.
2. Подать ток на динамик, установив бит 1 порта 61h.
3. Используя регистр ah, поместить нужное значение впорт 42h, определив тем самым нужную высоту тона.

Ниже в листинге 7.1 приведена программа, реализующая звук сирены. Многие команды вам уже знакомы, некоторые мы пока еще не рассматривали, поэтому поясним их функциональное назначение. Подробно они будут рассмотрены в свое время. Для удобства работы в программе была использована макрокоманда `delay`, выполняющая задержку работы программы на заданное время. Не стоит пока пытаться разбираться с механизмом макроподстановок — этим мы также займемся в свое время. Сейчас только отметьте для себя, что введенная таким образом макрокоманда в тексте программы синтаксически ничем не отличается от других команд ассемблера, и это позволяет программисту при необходимости расширить стандартный набор команд ассемблера. Введите текст макрокоманды `delay` (строки 11–25) и воспринимайте ее чисто по функциональному назначению (задержка выполнения программы на промежуток времени, задаваемый значением ее операнда). Стоит отметить, что данная макрокоманда чувствительна к производительности микропроцессора, из-за чего звуки на компьютерах с разными моделями микропроцессоров могут не совпадать. Сегмент кода, как обычно, начинается с настройки сегментного регистра ds на начало сегмента данных (строки 32–33). После этого строками 37–38 мы выполняем действия первого этапа — настройку канала 2, которая заключается в записи в регистр управления (порт 43h) байта состояния 0B6h. На втором шаге мы должны установить биты 0 и 1 порта 61h. Предварительно необходимо извлечь содержимое этого порта. Это делается для того, чтобы выполнять установку бит 0 и 1, не изменяя содержимого остальных бит порта 61h (строки 39–41). Принцип формирования сигнала сирены заключается в том, что в цикле на единицу изменяется содержимое регистра счетчика, и делается небольшая задержка для того, чтобы сигнал некоторое время звучал с нужной высотой. Постепенное повышение, а затем понижение высоты и дает нам эффект сирены. Строки 43–53 соответствуют циклу, в теле которого высота повышается, а строки 55–62 — циклу понижения тона. Оба цикла повторяются последовательно 5 раз. Контроль за этим осуществляется с помощью переменной `cnt`, содержимое которой увеличивается на 1 в строке 69 и контролируется на равенство 5 в строках 71–72. Если `cnt` = 5, то команда `cmp` устанавливает определенные флаги. Последующая команда условного перехода `jne` анализирует эти флаги и в зависимости от их состояния передает управление либо

на метку, указанную в качестве операнда этой команды, либо на следующую за нее команду. Цикл в программе ассемблера можно организовать несколькими способами; все они будут подробно разобраны на уроке 9. В данном случае цикл организуется командой `loop`, которая в качестве операнда имеет имя метки. На эту метку и передается управление при выполнении команды `loop`. Но до того как передать управление, команда `loop` анализирует содержимое регистра `esx/cx`, и, если оно равно нулю, управление передается не на метку, а на следующую за `loop` команду. Если содержимое регистра `esx/cx` не равно нулю, то оно уменьшается на единицу и управление передается на метку. Таким образом в `esx/cx` хранится *счетчик цикла*. В нашей программе он загружается в `esx/cx` в строках 42 и 54.

Листинг 7.1. Сирена

```
<1> :——Prg_7_1.asm——
<2> ;Программа, имитирующая звук сирены.
<3> ;Изменение высоты звука от 450 Гц до 2100 Гц.
<4> ;Используется макрос delay (задержка).
<5> ;При необходимости
<6> ;можно поменять значение задержки (по умолчанию – для процессора
      Pentium).
<7> ;
<8> masm
<9> model small
<10> stack 100h
<11> delay macro time
<12> ;макрос задержки, его текст ограничивается директивами macro и endm.
<13> ;На входе – значение задержки (в нкс)
<14>     local ext,iter
<15>     push cx
<16>     mov cx,time
<17> ext:
<18>     push cx
<19>     mov cx,5000
<20> iter:
<21>     loop iter
<22>     pop cx
<23>     loop ext
<24>     pop cx
<25> endm
<26> .data ;сегмент данных
<27> tonelow dw 2651 ;нижняя граница звучания 450 Гц
<28> cnt db 0 ;счетчик для выхода из программы
<29> temp dw ? ;верхняя граница звучания
<30> .code
<31> main: ;сегмент кода
<32>     mov ax,@data ;точка входа в программу
<33>     mov ds,ax ;связываем регистр ds с сегментом
<34>     mov ax,0 ;данных через регистр ax
<35> go: ;очищаем ax
<36>     ;заносим слово состояния 10110110b(0B6h) в командный регистр (порт 43h)
<37>     mov al,0B6h
```

```

<38>      out  43h,a1
<39>      in   a1,61h    ;получим значение порта 61h в a1
<40>      or   a1,3      ;инициализируем динамик и подаем ток в порт 61h
<41>      out  61h,a1
<42>      mov   cx,2083  ;количество шагов ступенчатого изменения тона
<43>      musicup:
<44>
<45>      mov   ax,tonelow
<46>      out  42h,a1    ;в порт 42h младшее слово ах:а1
<47>      xchg al,ah    ;обмен между a1 и ah
<48>      out  42h,a1    ;в порт 42h старшее слово ах:ах
<49>      add   tonelow,1 ;повышаем тон
<50>      delay 1       ;задержка на 1 мкс
<51>      mov   dx,tonelow ;в dx текущее значение высоты
<52>      mov   temp,dx  ;temp – верхнее значение высоты
<53>      loop  musicup ;повторить цикл повышения
<54>      mov   cx,2083  ;восстановить счетчик цикла
<55>      musicdown:
<56>      mov   ax,temp  ;в ах верхнее значение высоты
<57>      out  42h,a1    ;в порт 42h младшее слово ах:а1
<58>      mov   a1,ah    ;обмен между a1 и ah
<59>      out  42h,a1    ;в порт 42h старшее слово ах:ах
<60>      sub   temp,1   ;понижаем высоту
<61>      delay 1       ;задержка на 1 мкс
<62>      loop  musicdown ;повторить цикл понижения
<63>      nosound:
<64>      in   a1,61h    ;получим значение порта 61h в AL
<65>      and  a1,0FCh  ;выключить динамик
<66>      out  61h,a1
<67>      mov   dx,2651  ;для последующих циклов
<68>      mov   tonelow,dx
<69>      inc   cnt     ;увеличиваем счетчик проходов, то есть
<70>                  ;количество звучаний сирены
<71>      cmp   cnt,5   ;5 раз ?
<72>      jne   go      ;если нет, идти на метку go
<73>      exit:
<74>      mov   ax,4c00h  ;стандартный выход
<75>      int   21h
<76>      end main    ;конец программы

```

На диске в каталоге данного урока есть еще один пример использования команд ввода-вывода в порт — это редактор CMOS-памяти. На данном этапе изучения ассемблера не стоит пытаться разобраться с ним, но впоследствии стоит обязательно к нему вернуться.

Работа с адресами и указателями

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

1ea назначение, источник — загрузка эффективного адреса;

1ds назначение, источник — загрузка указателя в регистр сегмента данных **ds**;

1es назначение, источник — загрузка указателя в регистр дополнительного сегмента данных **es**;

1gs назначение, источник — загрузка указателя в регистр дополнительного сегмента данных **gs**;

1fs назначение, источник — загрузка указателя в регистр дополнительного сегмента данных **fs**;

1ss назначение, источник — загрузка указателя в регистр сегмента стека **ss**.

Команда **1ea** похожа на команду **mov** тем, что она также производит пересылку. Однако, обратите внимание, команда **1ea** производит пересылку не данных, а **эффективного адреса** данных (то есть смещения данных относительно начала сегмента данных) в регистр, указанный операндом **назначение**.

Часто для выполнения некоторых действий в программе недостаточно знать значение одного лишь эффективного адреса данных, а необходимо иметь полный указатель на данные. Вы помните, что полный указатель на данные состоит из сегментной составляющей и смещения. Все остальные команды этой группы позволяют получить в паре регистров такой полный указатель на операнд в памяти. При этом имя сегментного регистра, в который помещается сегментная составляющая адреса, определяется кодом операции. Соответственно, смещение помещается в регистр общего назначения, указанный операндом **назначение**. Но не все так просто с операндом **источник**. На самом деле в команде в качестве **источника** нельзя указывать непосредственно имя операнда в памяти, на который мы бы хотели получить указатель. Предварительно необходимо получить само значение полного указателя в некоторой области памяти и указать в команде получения полного адреса имя этой области. Для выполнения этого действия необходимо вспомнить директивы резервирования и инициализации памяти (см. урок 5). При применении этих директив возможен частный случай, когда в поле операндов указывается имя другой директивы определения данных (фактически, имя переменной). В этом случае в памяти формируется адрес этой переменной. Какой адрес будет сформирован (эффективный или полный), зависит от применяемой директивы. Если это **fw**, то в памяти формируется только 16-битное значение эффективного адреса, если же **dd** — в память записывается полный адрес. Размещение этого адреса в памяти следующее: в младшем слове находится смещение, в старшем — 16-битная сегментная составляющая адреса. Посмотрите на листинг 5.2 и рис. 5.18. В сегменте данных программы листинга 5.2 переменные **adr** и **adr_full** иллюстрируют наш случай получения частичного и полного указателей на данные в памяти.

Например, при организации работы с цепочкой символов удобно поместить ее начальный адрес в некоторый регистр и далее в цикле модифицировать это значение для последовательного доступа к элементам цепочки. В листинге 7.2 производится копирование строки байт **str_1** в строку байт **str_2**. В строках 12 и 13 в регистры **si** и **di** загружаются значения эффективных адресов переменных **str_1** и **str_2**. В строках 16, 17 производится пересылка очередного байта из одной строки в другую. Указатели на позиции байтов в строках определяются содержимым регистров **si** и **di**. Для пересылки очередного байта необходимо увеличить

на единицу регистры *si* и *di*, что и делается командами сложения *inc* (строки 18, 19). После этого программу необходимо зациклить до обработки всех символов строки.

Листинг 7.2. Копирование строки

```
<1> ;——Prg_7_2.asm——
<2> masm
<3> model small
<4> .data
<5> ...
<6> str_1 db "Ассемблер – базовый язык компьютера"
<7> str_2 db 50 dup (" ")
<8> full_pnt dd str_1
<9> ...
<10> .code
<11> start:
<12> ...
<13>     lea    si,str_1
<14>     lea    di,str_2
<15>     les    bx,full_pnt ;полный указатель на str1 в пару es:bx
<16> m1:
<17>     mov    al,[si]
<18>     mov    [di],al
<19>     inc    si
<20>     inc    di
<21> ;цикл на метку m1 до пересылки всех символов
<22> ...
<23> end start
```

Необходимость использования команд получения полного указателя данных в памяти, то есть адреса сегмента и значения смещения внутри сегмента, возникает, в частности, при работе с цепочками. Мы рассмотрим этот вопрос на уроке 11. В строке 8 листинга 7.2 в двойном слове *full_pnt* формируются сегментная часть адреса и смещение для переменной *str_1*. При этом 2 байта смещения занимают младшее слово *full_pnt*, а значение сегментной составляющей адреса – старшее слово *full_pnt*. В строке 15 командой *les* эти компоненты адреса помещаются в регистры *bx* и *es*.

Преобразование данных

К этой группе можно отнести множество команд микропроцессора, но большинство из них имеют те или иные особенности, которые требуют отнести их к другим функциональным группам (см. рис 6.10). Поэтому из всей совокупности команд микропроцессора непосредственно к командам преобразования данных можно отнести только одну команду:

xlat [адрес_таблицы_перекодировки]

Это очень интересная и полезная команда. Ее действие заключается в том, что она замещает значение в регистре *al* другим байтом из таблицы в памяти, расположенной по адресу, указанному операндом *адрес_таблицы_перекодировки*.

Слово «таблица» весьма условно; по сути, это просто строка байт. Адрес байта в строке, которым будет производиться замещение содержимого регистра *a1*, определяется суммой (*bx*) + (*a1*), то есть содержимое *a1* выполняет роль индекса в байтовом массиве.

При работе с командой *xlat* обратите внимание на следующий тонкий момент. Несмотря на то что в команде указывается адрес строки байтов, из которой должно быть извлечено новое значение, этот адрес должен быть предварительно загружен (например, с помощью команды *lea*) в регистр *bx*. Таким образом, операнд *адрес_таблицы_перекодировки* на самом деле не нужен (необязательность операнда показана заключением его в квадратные скобки). Что касается строки байтов (таблицы перекодировки), то она представляет собой область памяти размером от 1 до 255 байт (диапазон числа без знака в 8-битном регистре).

В качестве иллюстрации работы данной команды мы рассмотрим программу из листинга 3.1. Вы помните, что эта программа преобразовывала двузначное шестнадцатеричное число, вводимое с клавиатуры (то есть в символьном виде), в эквивалентное двоичное представление в регистре *a1*. Ниже (листинг 7.3) приведен вариант этой программы с использованием команды *xlat*.

Листинг 7.3. Использование таблицы перекодировки

```
<1> :——Prg_7_3.asm—————  
<2> ;Программа преобразования двузначного шестнадцатеричного числа  
<3> ;в двоичное представление с использованием команды xlat.  
<4> ;Вход: исходное шестнадцатеричное число; вводится с клавиатуры.  
<5> ;Выход: результат преобразования в регистре a1.  
<6> .data ;сегмент данных  
<7> message db "Введите две шестнадцатеричные цифры,$"  
<8> tabl db 48 dup (0),0,1,2,3,4,5,6,7,8,9, 8 dup (0),  
<9> db 0ah,0bh,0ch,odh,0eh,0fh,27 dup (0)  
<10> db 0ah,0bh,0ch,odh,0eh,0fh, 153 dup (0)  
<11> .stack 256 ;сегмент стека  
<12> .code  
<13> ;начало сегмента кода  
<14> proc main ;начало процедуры main  
<15>     mov ax,@data ;физический адрес сегмента данных в регистр ax  
<16>     mov ds,ax ;ах записываем в ds  
<17>     lea bx,tabl ;загрузка адреса строки байт в регистр bx  
<18>     mov ah,9  
<19>     mov dx,offset message  
<20>     int 21h ;вывести приглашение к вводу  
<21>     xor ax,ax ;очистить регистр ах  
<22>     mov ah,1h ;значение 1h в регистр ah  
<23>     int 21h ;вводим первую цифру в a1  
<24>     xlat ;перекодировка первого введенного символа в a1  
<25>     mov d1,a1  
<26>     shl d1,4 ;сдвиг d1 влево для освобождения места для младшей  
           ;цифры  
<27>     int 21h ;ввод второго символа в a1  
<28>     xlat ;перекодировка второго введенного символа в a1  
<29>     add a1,d1 ;складываем для получения результата  
<30>     mov ax,4c00h ;пересылка 4c00h в регистр ах  
<31>     int 21h ;завершение программы
```

```
<32> endp main      :конец процедуры main
<33> code ends      :конец сегмента кода
<34> end main        :конец программы с точкой входа main
```

Сама по себе программа проста; сложность вызывает обычно формирование таблицы перекодировки. Обсудим этот момент подробнее. Прежде всего нужно определиться со значениями тех байтов, которые вы будете изменять. В нашем случае это символы шестнадцатеричных цифр. На уроке 3 мы рассматривали их коды ASCII. Поэтому мы конструируем в сегменте данных таблицу, в которой на места байтов, соответствующих символам шестнадцатеричных цифр, помещаем их новые значения, то есть двоичные эквиваленты шестнадцатеричных цифр. Строки 8–10 листинга 7.3 демонстрируют, как это сделать. Байты этой таблицы, смещения которых не совпадают со значением кодов шестнадцатеричных цифр, нулевые. Таковыми являются первые 48 байт таблицы, промежуточные байты и часть в конце таблицы. Желательно определить все 256 байт таблицы. Дело в том, что если мы ошибочно поместим в `a1` код символа, отличный от символа шестнадцатеричной цифры, то после выполнения команды `xlat` получим непредсказуемый результат. В случае листинга 7.3 это будет ноль, что не совсем корректно, так как непонятно, что же в действительности было в `a1`: код символа «0» или что-то другое. Поэтому, наверное, есть смысл здесь поставить «защиту от дурака», поместив в неиспользуемые байты таблицы какой-нибудь определенный символ. После каждого выполнения `xlat` нужно будет просто контролировать значение в `a1` на предмет совпадения с этим символом, и если оно произошло, выдавать сообщение об ошибке.

После того как таблица составлена, с ней можно работать. В сегменте команд строка 17 инициализирует регистр `bx` значением адреса таблицы `tab1`. Далее все очень просто. Поочередно вводятся символы двух шестнадцатеричных цифр и производится их перекодировка в соответствующие двоичные эквиваленты. В остальном программа аналогична листингу 3.1.

Для закрепления знаний и исследования трудных моментов выполните программу из листинга 7.3 под управлением отладчика.

Работа со стеком

Эта группа представляет собой набор специализированных команд, ориентированных на организацию гибкой и эффективной работы со стеком. *Стек* – это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него в структуре программы предусмотрен отдельный сегмент. На тот случай, если программист забыл опи- сать сегмент стека в своей программе, компоновщик `tlink` выдаст предупреждаю- щее сообщение.

Для работы со стеком предназначены три регистра:

- `ss` – сегментный регистр стека;
- `sp/esp` – регистр указателя стека;
- `bp/ebp` – регистр указателя базы кадра стека.

Размер стека зависит от режима работы микропроцессора и ограничивается 64 Кбайт (или 4 Гбайт в защищенном режиме). В каждый момент времени доступен только один стек, адрес сегмента которого содержится в регистре ss. Этот стек называется *текущим*. Для того чтобы обратиться к другому стеку («переключить стек»), необходимо загрузить в регистр ss другой адрес. Регистр ss автоматически используется процессором для выполнения всех команд, работающих со стеком.

Перечислим еще некоторые особенности работы со стеком:

- Запись и чтение данных в стеке осуществляются в соответствии с принципом LIFO (Last In First Out — «последним пришел, первым ушел»).
- По мере записи данных в стек последний растет в сторону младших адресов. Эта особенность заложена в алгоритм работы со стеком.
- При использовании регистров esp/sp и ebp/bp для адресации памяти ассемблер автоматически считает, что содержащиеся в нем значения представляют собой смещения относительно сегментного регистра ss.

В общем случае стек организован так, как показано на рис. 7.3.

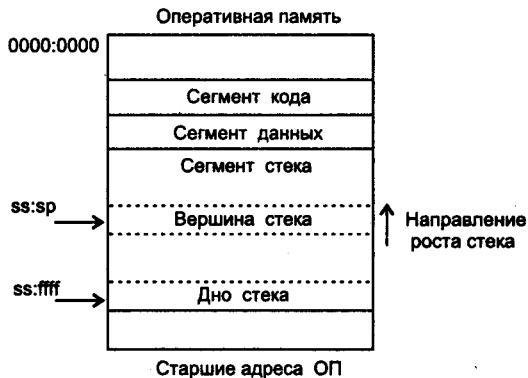


Рис. 7.3. Концептуальная схема организации стека

Регистры ss, esp/sp и ebp/bp¹ используются комплексно, и каждый из них имеет свое функциональное назначение. Регистр esp/sp всегда указывает на вершину стека, то есть содержит смещение, по которому в стек был занесен последний элемент. Команды работы со стеком неявно изменяют этот регистр так, чтобы он указывал всегда на последний записанный в стек элемент. Если стек пуст, то значение esp равно адресу последнего байта сегмента, выделенного под стек. При занесении элемента в стек процессор уменьшает значение регистра esp, а затем записывает элемент по адресу новой вершины. При извлечении данных из стека процессор копирует элемент, расположенный по адресу вершины, а затем увеличивает значение регистра указателя стека esp. Таким образом, получается, что стек растет вниз, в сторону уменьшения адресов.

¹ Какой из регистров применяется для адресации, 32-битный или 16-битный, зависит от значения модификатора use16 или use32 в директиве сегментации segment (см. урок 6).

Что делать, если нам необходимо получить доступ к элементам не на вершине, а внутри стека? Для этого применяют регистр `esp`. Регистр `esp` — регистр указателя базы кадра стека. Например, типичным приемом при входе в подпрограмму является передача нужных параметров путем записи их в стек. Если подпрограмма тоже активно работает со стеком, то доступ к этим параметрам становится проблематичным. Выход в том, чтобы после записи нужных данных в стек сохранить адрес вершины стека в *указателе кадра (базы) стека* — регистре `esp`. Значение в `esp` в дальнейшем можно использовать для доступа к переданным параметрам.

Начало стека расположено в старших адресах памяти. На рис. 7.3 этот адрес обозначен парой `ss:ffff`. Смещение `ffff` приведено здесь условно. Реально это значение определяется величиной, которую программист задает при описании сегмента стека в своей программе. К примеру, для программы в листинге 7.1 началу стека будет соответствовать пара `ss:0100h`. Адресная пара `ss:ffff` — это максимальное для реального режима значение адреса начала стека, так как размер сегмента в нем ограничен величиной 64 Кбайт (`0ffffh`).

Для организации работы со стеком существуют специальные команды записи и чтения.

`push источник` — запись значения *источник* в вершину стека.

Интерес представляет алгоритм работы этой команды, который включает следующие действия (рис. 7.4):

- $(sp) = (sp) - 2$; значение `sp` уменьшается на 2;
- значение из *источника* записывается по адресу, указанному парой `ss:sp`.

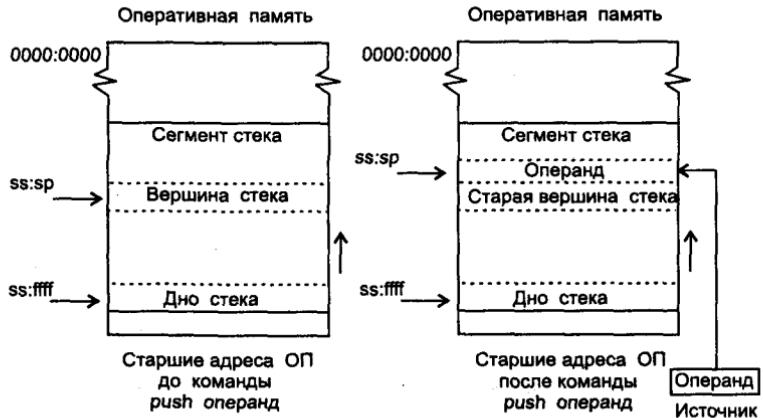


Рис. 7.4. Принцип работы команды `push`

`pop назначение` — запись значения из вершины стека по месту, указанному операндом *назначение*. Значение при этом «снимается» с вершины стека.

Алгоритм работы команды `pop` обратен алгоритму команды `push` (рис. 7.5):

- запись содержимого вершины стека по месту, указанному операндом *назначение*;
- $(sp) = (sp) + 2$; увеличение значения `sp`.

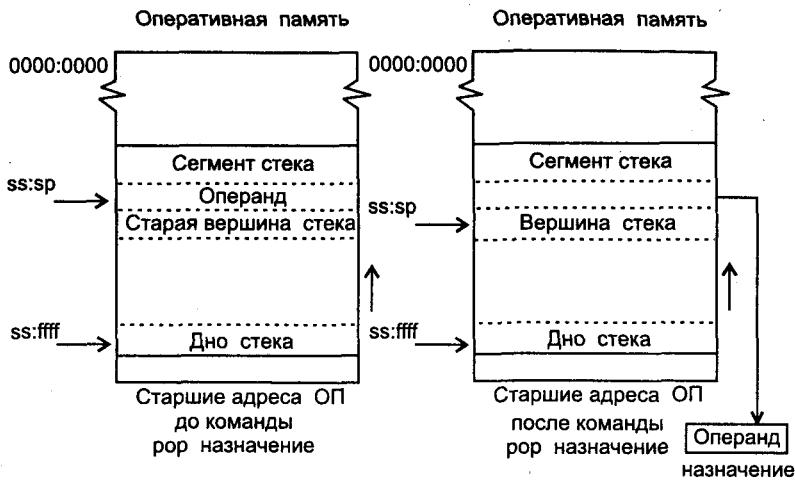


Рис. 7.5. Принцип работы команды pop

pusha – команда групповой записи в стек. По этой команде в стек последовательно записываются регистры ax, cx, dx, bx, sp, bp, si, di.

Заметим, что записывается оригинальное содержимое sp, то есть то, которое было до выдачи команды pusha (рис. 7.6).

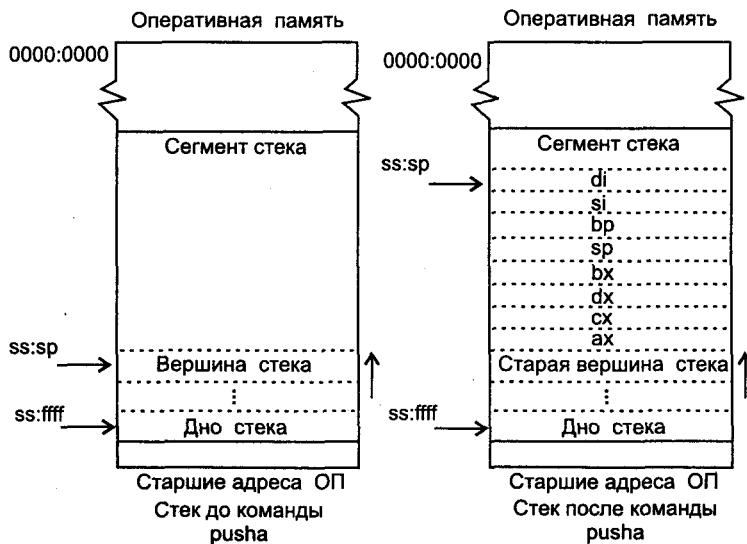


Рис. 7.6. Принцип работы команды pusha

pushaw – почти синоним команды pusha.

В чем разница? На уроке 5 мы обсуждали один из атрибутов сегмента – *атрибут разрядности*. Он может принимать значения use16 или use32. Рассмотрим работу команд pusha и pushaw при каждом из этих атрибутов.

- use16 — алгоритм работы pushaw аналогичен алгоритму pusha;
- use32 — pushaw не изменяется (то есть она нечувствительна к разрядности сегмента и всегда работает с регистрами размером в слово — ax, cx, dx, bx, sp, bp, si, di). Команда pusha чувствительна к установленной разрядности сегмента и при указании 32-разрядного сегмента работает с соответствующими 32-разрядными регистрами, то есть eax, ecx, edx, ebx, esp, ebp, esi, edi.

pushad — выполняется аналогично команде pusha, но есть некоторые особенности, которые вы можете найти в Справочнике.

Следующие три команды выполняют действия, обратные вышеописанным командам:

- popa;
- popaw;
- popad.

Группа команд, описанная ниже, позволяет сохранить в стеке регистр флагов и записать слово или двойное слово в стеке. Отметим, что перечисленные ниже команды — единственные в системе команд микропроцессора, которые позволяют получить доступ (и которые нуждаются в этом доступе) ко всему содержимому регистра флагов.

pushf — сохраняет регистр флагов в стеке.

Работа этой команды зависит от атрибута размера сегмента:

use16 — в стек записывается регистр flags размером 2 байта;

use32 — в стек записывается регистр eflags размером 4 байта;

pushfw — сохраняет в стеке регистр флагов размером в слово. Всегда работает как pushf с атрибутом use16;

pushfd — сохраняет в стеке регистр флагов flags или eflags, в зависимости от атрибута разрядности сегмента (то есть то же, что и pushf).

Аналогично, следующие три команды выполняют действия, обратные рассмотренным выше операциям:

- popf;
- popfw;
- popfd.

Работать со стеком приходится постоянно, поэтому к этому вопросу мы будем возвращаться еще не раз. Отметим основные виды операций, когда использование стека практически неизбежно:

- вызов подпрограмм;
- временное сохранение значений регистров;
- определение локальных переменных.

Подведем некоторые итоги:

- Основная команда пересылки данных — `mov`. Операнды этой команды (как и большинства других команд, берущих значения из памяти) должны быть согласованы по разрядности. Несмотря на то что в некоторых случаях действуют правила умолчания, в сомнительных ситуациях лучше явно указывать разрядность operandов с помощью оператора `ptr`.
- Управление периферией компьютера осуществляется, в общем случае, с использованием всего двух команд ввода-вывода, `in` и `out`.
- В процессе работы программы динамически можно получить как эффективный, так и полный (физический) адрес памяти. Для этого язык ассемблера предоставляет группу команд получения указателей памяти.
- Архитектура микропроцессора предоставляет в распоряжение программиста специфическую, но весьма эффективную структуру — стек. Система команд поддерживает все необходимые операции со стеком. Подробнее со стеком мы познакомимся при изучении модульного программирования на ассемблере.

8

УРОК

Арифметические команды

-
- Форматы арифметических данных
 - Арифметические операции над двоичными числами
 - Арифметические операции над десятичными (BCD) числами
-

На уроке 1 при обсуждении истории компьютеров мы упомянули причину, побудившую человека искать себе помощника. Вспомнили? Совершенно верно, — это было желание эффективно решать счетные задачи. Правда, путь к собственно эффективному решению оказался несколько дольше, чем хотелось бы, но, тем не менее, именно благодаря этому стремлению человечество имеет сегодня определенные достижения. Любой компьютер, от самого примитивного до супермощного, имеет в своей системе команды для выполнения арифметических действий. Работая с компьютером при помощи языков высокого уровня, мы воспринимаем возможность проведения расчетных действий как должное, забывая при этом, что компилятор даже очень развитого языка программирования превращает все самые высокоуровневые действия в унылую последовательность машинных команд. Конечно, мало кому придет в голову писать серьезную расчетную задачу на ассемблере. Но даже в системных программах часто требуется проведение небольших вычислений. Поэтому важно разобраться с этой группой команд. К тому же, она на удивление очень компактна и не избыточна.

Микропроцессор может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных блока:

- устройство для выполнения целочисленных операций;
- устройство для выполнения операций с плавающей точкой.

Каждое из этих устройств имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Устройство с плавающей точкой и его система команд будут рассмотрены на уроке 19. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.

Общий обзор

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд. На рис. 8.1 приведена классификация команд этой группы.

Группа арифметических целочисленных команд работает с двумя типами чисел:

- целыми двоичными числами. Числа могут иметь знаковый разряд или не иметь такового, то есть быть числами со знаком или без знака;
- целыми десятичными числами.

На уроке 2 мы обсуждали вопрос о данных, с которыми работают арифметические команды. Вспомним некоторые ключевые моменты.

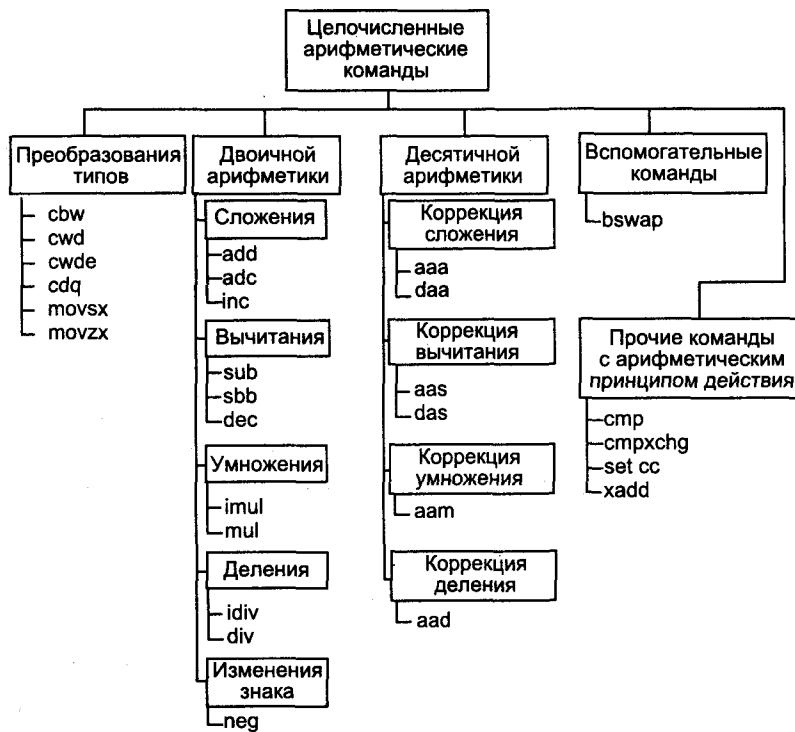


Рис. 8.1. Классификация арифметических команд

Целые двоичные числа

Целое двоичное число с фиксированной точкой — это число, закодированное в двоичной системе счисления. Размерность целого двоичного числа может составлять 8, 16 или 32 бита. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа. Это 7-й, 15-й или 31-й биты для чисел соответствующей размерности (см. урок 2). При этом интересно то, что среди арифметических команд есть всего две команды, которые действительно учитывают этот старший разряд как знаковый, — это команды целочисленного умножения и деления *imul* и *idiv*. В остальных случаях ответственность за действия со знаковыми числами и, соответственно, со знаковым разрядом ложится на программиста. К этому вопросу мы вернемся чуть позже. Диапазон значений двоичного числа зависит от его размера и трактовки старшего бита либо как старшего значащего бита числа, либо как бита знака числа (табл. 8.1).

Как описать числа с фиксированной точкой в программе? Это делается с использованием директив описания данных *db*, *dw* и *dd*, рассмотренных на уроке 6. В Справочнике описаны возможные варианты содержимого полей операндов этих директив и диапазоны их значений. К примеру, последовательность описаний дво-

иных чисел из сегмента данных листинга 8.1 (помните о принципе «младший байт по младшему адресу») будет выглядеть в памяти так, как показано на рис. 8.2.

Таблица 8.1. Диапазон значений двоичных чисел

| Размерность поля | Целое без знака | Целое со знаком |
|------------------|------------------|---------------------------------|
| Байт | 0...255 | -128...+127 |
| Слово | 0...65 535 | -32 768...+32 767 |
| Двойное слово | 0..4 294 967 295 | -2 147 483 648...+2 147 483 647 |

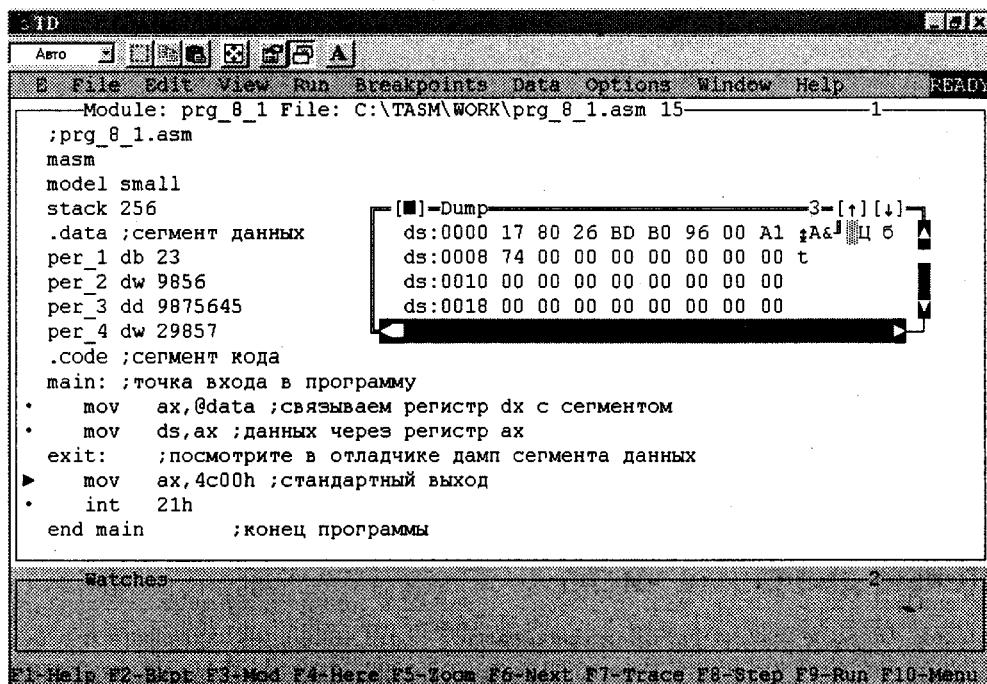


Рис. 8.2. Дамп памяти для сегмента данных листинга 8.1

Листинг 8.1. Числа с фиксированной точкой

```
:prg_8_1.asm
masm
model small
stack 256
.data ;сегмент данных
per_1 db 23
per_2 dw 9856
per_3 dd 9875645
per_4 dw 29857
.code ;сегмент кода
main: ;точка входа в программу
```

```

mov ax, @data      ;связываем регистр dx с сегментом
mov ds, ax         ;данных через регистр ax
exit:             ;посмотрите в отладчике данн сегмента данных
    mov ax, 4c00h   ;стандартный выход
    int 21h
end main          ;конец программы

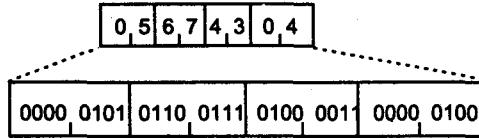
```

Десятичные числа

Десятичные числа — специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех бит. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом *двоично-десятичном коде* (BCD — Binary-Coded Decimal). Микропроцессор хранит BCD-числа в двух форматах (рис. 8.3):

- *упакованный формат* — каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером 4 бита. При этом код старшей цифры числа занимает старшие 4 бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99;
- *неупакованный формат* — каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие четыре бита имеют нулевое значение. Это так называемая *зона*. Следовательно, диапазон представления десятичного неупакованного числа в одном байте составляет от 0 до 9.

Упакованное десятичное число 5674304:



Неупакованное десятичное число 9985784:

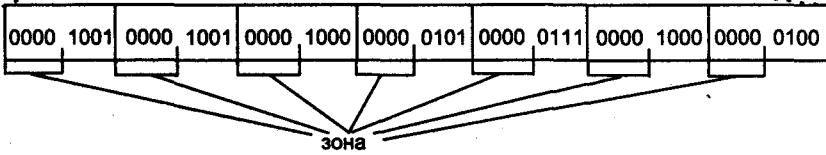


Рис. 8.3. Представление BCD-чисел

Как описать двоично-десятичные числа в программе? Для этого можно использовать только две директивы описания и инициализации данных — db и dt. Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип «младший байт по младшему адресу», что, как мы увидим далее, очень удобно для их обработки.

И вообще, при использовании такого типа данных, как BCD-числа, порядок описания этих чисел в программе и алгоритм их обработки — это дело вкуса и личных пристрастий программиста. Это станет более ясным после того, как мы ниже рассмотрим основы работы с BCD-числами. К примеру, приведенная в сегменте данных листинга 8.2 последовательность описаний BCD-чисел будет выглядеть в памяти так, как показано на рис. 8.4.

Листинг 8.2. BCD-числа

```
:prg_8_2.asm
masm
model small
stack 256
.data
per_1 db 2,3,4,6,8,2      ;сегмент данных
per_3 dt 9875645          ;неупакованное BCD-число 286432
                           ;упакованное BCD-число 9875645
.code
main:
    mov ax,@data           ;связываем регистр dx с сегментом
    mov ds,ax               ;данных через регистр ax
exit:
    mov ax,4c00h            ;посмотрите в отладчике дамп сегмента данных
    int 21h                 ;стандартный выход
end main                  ;конец программы
```

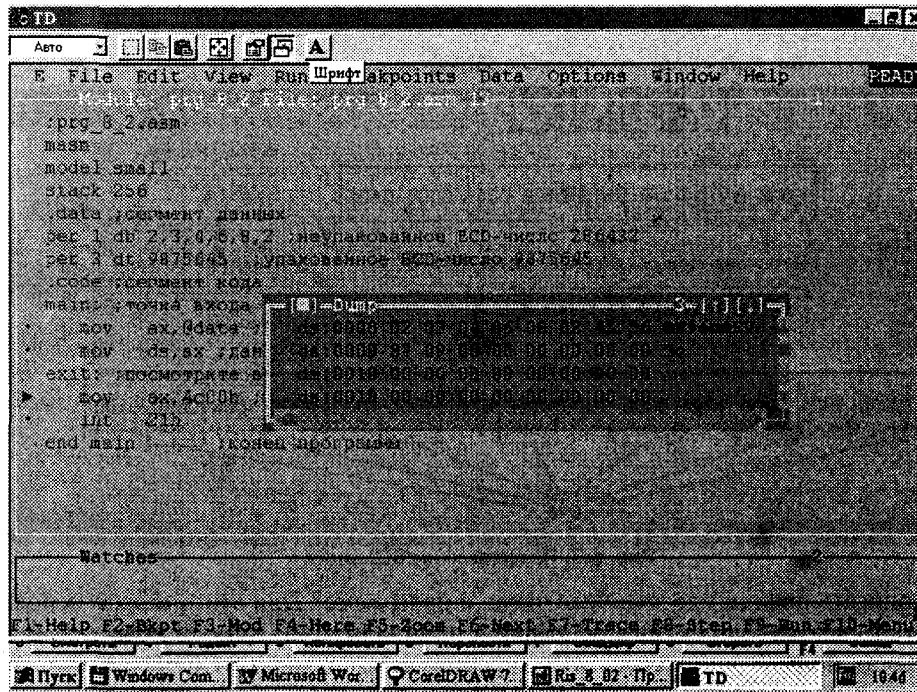


Рис. 8.4. Дамп памяти для сегмента данных листинга 8.2

После столь подробного обсуждения объектов, с которыми работают арифметические операции, можно приступить к рассмотрению средств их обработки на уровне системы команд микропроцессора.

Арифметические операции над целыми двоичными числами

В данном разделе мы рассмотрим особенности каждого из четырех основных арифметических действий для двоичных чисел со знаком и без знака.

Сложение двоичных чисел без знака

Микропроцессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда (см. табл. 8.1). Например, при сложении операндов размером в байт результат не должен превышать число 255. Если это происходит, то результат оказывается неверен. Рассмотрим, почему так происходит. К примеру, выполним сложение: $254 + 5 = 259$ в двоичном виде. $11111110 + 0000101 = 1\ 00000011$. Результат вышел за пределы восьми бит, и правильное его значение укладывается в 9 битов, а в 8-битном поле операнда осталось значение 3, что, конечно, неверно. В микропроцессоре этот исход сложения прогнозируется, и предусмотрены специальные средства для фиксирования подобных ситуаций и их обработки. Так, для фиксирования ситуации выхода за разрядную сетку результата, как в данном случае, предназначен *флаг переноса cf*. Он располагается в бите 0 регистра флагов *eFlags/flags*. Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Естественно, что программист должен предусматривать возможность такого исхода операции сложения и средства для корректировки. Это предполагает включение участков кода после операции сложения, в которых анализируется флаг *cf*. Анализ этого флага можно провести различными способами. Самый простой и доступный — использовать команду условного перехода *jc*. Эта команда в качестве операнда имеет имя метки в текущем сегменте кода. Переход на эту метку осуществляется в случае, если в результате работы предыдущей команды флаг *cf* установленлся в 1. Команды условных переходов будут рассматриваться на уровне 10.

Если теперь посмотреть на рис. 8.1, то видно, что в системе команд микропроцессора имеются три команды двоичного сложения:

- *inc операнд* — операция *инкремента*, то есть увеличения значения операнда на 1;
- *add операнд_1,операнд_2* — команда сложения с принципом действия: *операнд_1 = операнд_1 + операнд_2*;
- *adc операнд_1,операнд_2* — команда сложения с учетом флага переноса *cf*.

Принцип действия команды:

операнд_1 = операнд_1 + операнд_2 + значение_cf

Обратите внимание на последнюю команду — это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления такой единицы мы уже рассмотрели. Таким образом, команда adc является средством микропроцессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые микропроцессором длины стандартных полей.

Рассмотрим пример вычисления суммы чисел (листинг 8.3).

Листинг 8.3. Вычисление суммы чисел

```
<1> :prg_8_3.asm
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> a db 254
<7> .code ;сегмент кода
<8> main:
<9>     mov ax,@data
<10>    mov ds,ax
<11> ...
<12>    xor ax,ax
<13>    add al,17
<14>    add al,a
<15>    jnc m1 ;если нет переноса, то перейти на m1
<16>    adc ah,0 ;в ах сумма с учетом переноса
<17> m1: ...
<18>    exit:
<19>    mov ax,4c00h ;стандартный выход
<20>    int 21h
<21> end main ;конец программы
```

В листинге 8.3 в строках 13–14 создана ситуация, когда результат сложения выходит за границы операнда. Эта возможность учитывается строкой 15, где команда jnc (хотя можно было обойтись и без нее) проверяет состояние флага cf. Если он установлен в 1, то это признак того, что результат операции получился больше по размеру, чем размер операнда, и для его корректировки необходимо выполнить некоторые действия. В данном случае мы просто полагаем, что границы операнда расширяются до размера ах, для чего учитываем перенос в старший разряд командой adc (строка 16). Если у вас остались вопросы, исследуйте работу команд сложения без учета знака, для чего введите листинг 8.3, получите исполняемый модуль, запустите отладчик и откройте в нем окна View ▶ Dump и View ▶ Registers.

Сложение двоичных чисел со знаком

Теперь настала пора раскрыть небольшой секрет. Дело в том, что на самом деле микропроцессор не подозревает о различии между числами со знаком и без знака. Вместо этого у него есть средства фиксирования возникновения характерных си-

туаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели при обсуждении сложения чисел без знака — это флаг переноса `cf`, установка которого в 1 говорит о том, что произошел выход за пределы разрядности операндов, и команда `adc`, которая учитывает возможность такого выхода (перенос из младшего разряда). Другое средство — это регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения `of` в регистре `eflags` (бит 11).

Из материала урока 6 вы помните, как представляются числа в компьютере: положительные числа — в двоичном коде, а отрицательные — в дополнительном коде. Рассмотрим различные варианты сложения чисел. Примеры призваны показать поведение двух старших битов операндов и правильность результата операции сложения.

Пример 8.1. Сложение чисел 1

$$\begin{array}{r} 30566 = 01110111 \ 01100110 \\ + \\ 00687 = 00000010 \ 10101111 \\ = \\ 31253 = 01111010 \ 00010101 \end{array}$$

Следим за переносами из 14-го и 15-го разрядов и правильностью результата: переносов нет, результат правильный.

Пример 8.2. Сложение чисел 2

$$\begin{array}{r} 30566 = 01110111 \ 01100110 \\ + \\ 30566 = 01110111 \ 01100110 \\ = \\ 61132 = 11101110 \ 11001100 \end{array}$$

Произошёл перенос из 14-го разряда; из 15-го разряда переноса нет. Результат неправильный, так как имеется *переполнение* — значение числа получилось больше, чем то, которое может иметь 16-битное число со знаком (+32 767).

Пример 8.3. Сложение чисел 3

$$\begin{array}{r} -30566 = 10001000 \ 10011010 \\ + \\ -04875 = 11101100 \ 11110101 \\ = \\ -35441 = 01110101 \ 10001111 \end{array}$$

Произошел перенос из 15-го разряда, из 14-го разряда нет переноса. Результат неправильный, так как вместо отрицательного числа получилось положительное (в старшем бите находится 0).

Пример 8.4. Сложение чисел 4

$-4875 = 11101100 \ 11110101$

+

$-4875 = 11101100 \ 11110101$

=

$-9750 = 11011001 \ 11101010$

Есть переносы из 14-го и 15-го разрядов. Результат правильный.

Таким образом, мы исследовали все случаи и выяснили, что ситуация *переполнения* (установка флага of в 1) происходит при переносе:

- из 14-го разряда (для положительных чисел со знаком);
- из 15-го разряда (для отрицательных чисел).

И наоборот, переполнения не происходит (то есть флаг of сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах.

Итак, переполнение регистрируется с помощью флага переполнения of. Дополнительно к флагу of при переносе из старшего разряда устанавливается в 1 и флаг переноса cf. Так как микропроцессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Теперь, наверное, понятно, почему мы столько внимания уделили тонкостям сложения чисел со знаком. Учтя все это, мы сможем организовать правильный процесс сложения чисел — будем анализировать флаги cf и of и принимать правильное решение! Проанализировать флаги cf и of можно командами условного перехода jc\jnc и jo\jno, соответственно.

Что же касается команд сложения чисел со знаком, то вы уже, наверное, догадались, что сами команды сложения чисел со знаком те же, что и для чисел без знака.

Вычитание двоичных чисел без знака

Как и при анализе операции сложения, порассуждаем над сутью процессов, происходящих при выполнении операции вычитания:

- Если уменьшаемое больше вычитаемого, то проблем нет, — разность положительна, результат верен.
- Если уменьшаемое меньше вычитаемого, возникает проблема: результат меньше 0, а это уже число со знаком. В этом случае результат необходимо *запомнить*. Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Микропроцессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим, в разрядной сетке операнда. Поясним на примере.

Пример 8.5. Вычитание чисел 1

$05 = 00000000 \ 00000101$

$-10 = 00000000 \ 00001010$

Для того чтобы произвести вычитание, произведем воображаемый заем из старшего разряда:

1 00000000 00000101

00000000 00001010

= 11111111 11110111

Тем самым, по сути, выполняется действие $(65\ 536 + 5) - 10 = 65\ 531$, 0 здесь как бы эквивалентен числу 65 536. Результат, конечно, неверен, но микропроцессор считает, что все нормально, хотя факт заема единицы он фиксирует установкой флага переноса cf. Но посмотрите еще раз внимательно на результат операции вычитания. Это же -5 в дополнительном коде! Проведем эксперимент: представим разность в виде суммы $5 + (-10)$.

Пример 8.6. Вычитание чисел 2

5 = 00000000 00000101

+
 $(-10) = 11111111 11110110$

= 11111111 11110111,

то есть мы получили тот же результат, что и в предыдущем примере.

Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага cf. Если он установлен в 1, то это говорит о том, что произошел заем из старшего разряда, и результат получился в дополнительном коде.

Аналогично командам сложения, группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматриваем, а учет особых ситуаций должен производиться самим программистом. К командам вычитания относятся следующие:

- dec *операнд* — операция декремента, то есть уменьшения значения операнда на 1;
- sub *операнд_1,операнд_2* — команда вычитания; ее принцип действия:
 операнд_1 = *операнд_1* - *операнд_2*;
- sbb *операнд_1,операнд_2* — команда вычитания с учетом заема (флага cf):
 операнд_1 = *операнд_1* - *операнд_2* - значение cf.

Как видите, среди команд вычитания есть команда sbb, учитывающая флаг переноса cf. Эта команда подобна adc, но теперь уже флаг cf выполняет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Рассмотрим пример (листинг 8.4) программной обработки ситуации, разобранной в примере 8.6.

Листинг 8.4. Проверка при вычитании чисел без знака

```
<1> :prg_8_4.asm
<2> masm
<3> model small
```

```

<4> stack 256
<5> .data
<6> .code      ;сегмент кода
<7> main:      ;точка входа в программу
<8> ...
<9>     xor  ax,ax
<10>    mov   al,5
<11>    sub   al,10
<12>    jnc   m1          ;нет переноса?
<13>    neg   al          ;в al модуль результата
<14> m1: ...
<15>    exit:
<16>    mov   ax,4c00h    ;стандартный выход
<17>    int   21h
<18> end main           ;конец программы

```

В этом примере в строке 11 выполняется вычитание. С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальному виду (получить его модуль), применяется команда neg, с помощью которой получается дополнение операнда. В нашем случае мы получили дополнение, или модуль отрицательного результата. А тот факт, что это на самом деле число отрицательное, отражен в состоянии флага cf. Дальше все зависит от алгоритма обработки. Исследуйте программу в отладчике.

Вычитание двоичных чисел со знаком

Здесь все несколько сложнее. Последний пример показал то, что микропроцессору незачем иметь два устройства — сложения и вычитания. Достаточно наличия только одного — устройства сложения. Но для вычитания способом сложения чисел со знаком в дополнительном коде необходимо представлять оба операнда — и уменьшаемое, и вычитаемое. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего, они связаны с тем, что старший бит операнда рассматривается как знаковый. Рассмотрим пример вычитания 45 — (-127).

Пример 8.7. Вычитание чисел со знаком 1

$$\begin{array}{r}
 45 = 0010\ 1101 \\
 - \\
 -127 = 1000\ 0001 \\
 = \\
 -44 = 1010\ 1100
 \end{array}$$

Судя по знаковому разряду, результат получился отрицательный, что, в свою очередь, говорит о том, что число нужно рассматривать как дополнение, равное -44. Правильный результат должен быть равен 172. Здесь мы, как и в случае знакового сложения, встретились с *переполнением мантиссы*, когда значащий разряд числа изменил знаковый разряд операнда. Отследить такую ситуацию можно

по содержимому флага переполнения of. Его установка в 1 говорит о том, что результат вышел за диапазон представления знаковых чисел (то есть изменился старший бит) для операнда данного размера, и программист должен предусмотреть действия по корректировке результата.

Другой пример разности рассматривается в примере 8.8, но выполним мы ее способом сложения.

Пример 8.8. Вычитание чисел со знаком 2

$$-45 - 45 = -45 + (-45) = -90.$$

$$\begin{array}{r} -45 \\ + (-45) \\ \hline -90 \end{array}$$

+

$$\begin{array}{r} -45 \\ + (-45) \\ \hline -90 \end{array}$$

=

$$\begin{array}{r} -90 \\ + (-45) \\ \hline -135 \end{array}$$

Здесь все нормально, флаг переполнения of сброшен в 0, а 1 в знаковом разряде говорит о том, что значение результата — число в дополнительном коде.

Вычитание и сложение operandов большой размерности

Если вы заметили, команды сложения и вычитания работают с operandами фиксированной размерности: 8, 16, 32 бит. А что делать, если нужно сложить числа большей размерности, например 48 бит, используя 16-разрядные operandы? К примеру, сложим два 48-разрядных числа (рис. 8.5):

| | | | |
|--|--|-------------------|------------------|
| 1 слагаемое | 0010001110010101 | 01001011 11111000 | 1111111100110001 |
| 2 слагаемое | + 0100010010001011 | 1010010100100100 | 0100100110000110 |
| 1 шаг : сложение младших 16 бит | $\begin{array}{r} 0010001110010101 \\ + 0100010010001011 \\ \hline 1010010010101011 \end{array}$ | | |
| 2 шаг : сложение средних 16 бит (с учетом переноса из младшего разряда): | $\begin{array}{r} 01001011 11111000 \\ + 1010010100100100 \\ \hline 11110001 00011100 \end{array}$ | | |
| 3 шаг : сложение старших 16 бит (переноса из младшего разряда нет): | $\begin{array}{r} 11110001 00011101 \\ + 0010001110010101 \\ \hline 0100100001000000 \end{array}$ | | |
| Результат сложения: | 0100100001000000 | 11110001 00011101 | 0100100010110111 |

Рис. 8.5. Сложение operandов большой размерности

На рис. 8.5 по шагам показана технология сложения длинных чисел. Видно, что процесс сложения многобайтных чисел происходит так же, как и при сложении двух чисел «в столбик», — с осуществлением при необходимости переноса 1

в старший разряд. Если нам удастся запрограммировать этот процесс, то мы значительно расширим диапазон двоичных чисел, над которыми мы сможем выполнять операции сложения и вычитания.

Принцип вычитания чисел с диапазоном представления, превышающим стандартные разрядные сетки операндов, тот же, что и при сложении, то есть используется флаг переноса cf. Нужно только представлять себе процесс вычитания в столбик и правильно комбинировать команды микропроцессора с командой sbb. Чтобы написать достаточно интересную программу, моделирующую этот процесс, необходимо привлечь те конструкции языка ассемблера, которые мы еще не обсуждали. На прилагаемой к книге дискете в каталоге данного урока находятся исходные тексты подпрограмм, реализующих четыре основных арифметических действия для двоичных операндов произвольной размерности. Не поленитесь внимательно изучить их, так как они являются хорошей иллюстрацией к материалу, изучаемому на этом и последующих уроках. К этим примерам можно будет обратиться в полной мере после того, как будут изучены механизмы процедур и макрокоманд (уроки 10 и 13).

В завершение обсуждения команд сложения и вычитания отметим, что кроме флагов cf и of в регистре eflags есть еще несколько флагов, которые можно использовать с двоичными арифметическими командами. Речь идет о следующих флагах:

- zf — флаг нуля, который устанавливается в 1, если результат операции равен 0, и в 0, если результат не равен 0;
- sf — флаг знака, значение которого после арифметических операций (и не только) совпадает со значением старшего бита результата, то есть с битом 7, 15 или 31. Таким образом, этот флаг можно использовать для операций над числами со знаком.

Умножение двоичных чисел без знака

Для умножения чисел без знака предназначена команда

`mul сомножитель_1`

Как видите, в команде указан всего лишь один операнд-сомножитель. Второй операнд-сомножитель_2 задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и размещения второго операнда и результата приведены в табл. 8.2.

Из таблицы видно, что произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах — на месте сомножитель_2 (младшая часть) и в дополнительных регистрах ah, dx, edx (старшая часть). Как же динамически (то есть во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре, или что он превысил размерность

регистра, и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам по предыдущему обсуждению флаги переноса cf и переполнения of:

- если старшая часть результата нулевая, то после операции произведения флаги cf = 0 и of = 0;
- если же эти флаги ненулевые, то это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Таблица 8.2. Расположение operandов и результата при умножении

| сомножитель_1 | сомножитель_2 | Результат |
|---------------|---------------|---|
| Байт | al | 16 бит в ах: al — младшая часть результата; ah — старшая часть результата |
| Слово | ax | 32 бит в паре dx:ax: ax — младшая часть результата; dx — старшая часть результата |
| Двойное слово | eax | 64 бит в паре edx:eax: eax — младшая часть результата; edx — старшая часть результата |

Рассмотрим следующий пример программы (листинг 8.5).

Листинг 8.5. Умножение

```
<1> :prg_8_5.asm
<2> masm
<3> model small
<4> stack 256
<5> .data ;сегмент данных
<6> rez label word
<7> rez_l db 45
<8> rez_h db 0
<9> .code ;сегмент кода
<10> main: ;точка входа в программу
<11> ...
<12> xor ax,ax
<13> mov a1,25
<14> mul rez_l ;если нет переполнения, то на m1
<15> jnc m1 ;старшую часть результата в rez_h
<16> mov rez_h,ah
<17> m1:
<18> mov rez_l,a1
<19> exit:
<20> mov ax,4c00h ;стандартный выход
<21> int 21h
<22> end main ;конец программы
```

В этой программе в строке 14 производится умножение значения в rez_l на число в регистре a1. Согласно информации в табл. 8.2, результат умножения будет располагаться в регистре a1 (младшая часть) и в регистре ah (старшая часть). Для выяснения размера результата в строке 15 командой условного перехода jnc

анализируется состояние флага cf, и если оно не равно 1, то результат остается в рамках регистра al. Если же cf = 1, то выполняется команда в строке 16, которая формирует в поле rez_h старшее слово результата. Команда в строке 18 формирует младшую часть результата. Теперь обратите внимание на сегмент данных, а именно на строку 6. В этой строке содержится директива label. Мы еще не раз будем сталкиваться с этой директивой. В данном случае она назначает еще одно символическое имя rez адресу, на который уже указывает другой идентификатор rez_1. Отличие заключается в типах этих идентификаторов — имя rez имеет тип слова, который ему назначается директивой label (имя типа указано в качестве операнда label). Введя эту директиву в программе, мы подготовились к тому, что, возможно, результат операции умножения будет занимать слово в памяти. Обратите внимание, что мы не нарушили принципа: младший байт по младшему адресу. Далее, используя имя rez, можно обращаться к значению в этой области как к слову.

В заключение вам осталось исследовать в отладчике программу на разных наборах сомножителей.

Умножение двоичных чисел со знаком

Для умножения чисел со знаком предназначена команда

imul [операнд_1,]операнд_2,операнд_3]

Эта команда выполняется так же, как и команда mul. Отличительной особенностью команды imul является только формирование знака. Если результат мал и умещается в одном регистре (то есть если cf = of = 0), то содержимое другого регистра (старшей части) является расширением знака — все его биты равны старшему биту (знаковому разряду) младшей части результата. В противном случае (если cf = of = 1) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата. Если вы найдете в Справочнике команду imul, то увидите, что она допускает более широкие возможности по заданию местоположения operandов. Это сделано для удобства использования.

Деление двоичных чисел без знака

Для деления чисел без знака предназначена команда

div делитель

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бит. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера operandов. Результатом команды деления являются значения частного и остатка. Варианты местоположения и размеров operandов операции деления показаны в табл. 8.3.

После выполнения команды деления содержимое флагов неопределенно, но возможно возникновение прерывания с номером 0, называемого «деление на ноль». Этот вид прерывания относится к так называемым *исключениям*. Эта разновидность прерываний возникает внутри микропроцессора из-за некоторых аномалий

во время вычислительного процесса. К вопросу об исключениях мы еще вернемся. Прерывание 0 — «деление на ноль» — при выполнении команды `div` может возникнуть по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную под него разрядную сетку, что может случиться в следующих случаях:
 - при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого в более чем 256 раз больше значения делителя;
 - при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого в более чем 65 536 раз больше значения делителя;
 - при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого в более чем 4 294 967 296 раз больше значения делителя.

Таблица 8.3. Расположение operandов и результата при делении

| Делимое | Делитель | Частное | Остаток |
|---|---|---|---|
| Слово 16 бит в регистре ax | Байт — регистр или ячейка памяти | Байт в регистре al | Байт в регистре ah |
| 32 бит dx — старшая часть ax — младшая часть | 16 бит регистр или ячейка памяти | Слово 16 бит в регистре ax | Слово 16 бит в регистре dx |
| 64 бит edx — старшая часть eax — младшая часть | Двойное слово 32 бит регистр или ячейка памяти | Двойное слово 32 бит в регистре eax | Двойное слово 32 бит в регистре edx |

К примеру, выполним деление значения в области `del` на значение в области `delt` (листинг 8.6).

Листинг 8.6. Деление чисел

```
<1> :prg_8.6.asm
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> del_b label byte
<7> del dw 29876
<8> delt db 45
<9> .code ;сегмент кода
<10> main: ;точка входа в программу
<11> ...
<12> xor ax,ax
<13> ;последующие две команды можно заменить одной mov ax,del
<14> mov ah,del_b ;старший байт делимого в ah
<15> mov al,del_b+1 ;младший байт делимого в al
```

```

<16>      div   delt      ; в a1 - частное, в ah - остаток
<17> ...
<18> end   main           ; конец программы

```

Деление двоичных чисел со знаком

Для деления чисел со знаком предназначена команда

idiv делитель

Для этой команды справедливы все рассмотренные положения, касающиеся команд и чисел со знаком. Отметим лишь особенности возникновения исключения 0 «деление на ноль» в случае чисел со знаком. Оно возникает при выполнении команды **idiv** по одной из следующих причин:

- **делитель** равен нулю;
- частное не входит в отведенную для него разрядную сетку. Последнее, в свою очередь, может произойти:
 - при делении делимого величиной в слово со знаком на **делитель** величиной в байт со знаком, причем значение делимого в более чем 128 раз больше значения **делителя** (таким образом, частное не должно находиться вне диапазона от -128 до +127);
 - при делении делимого величиной в двойное слово со знаком на **делитель** величиной в слово со знаком, причем значение делимого в более чем 32 768 раз больше значения **делителя** (таким образом, частное не должно находиться вне диапазона от -32 768 до +32 768);
 - при делении делимого величиной в учетверенное слово со знаком на **делитель** величиной в двойное слово со знаком, причем значение делимого в более чем 2 147 483 648 раз больше значения **делителя** (таким образом, частное не должно находиться вне диапазона от -2 147 483 648 до +2 147 483 647).

Вспомогательные команды для целочисленных операций

В системе команд микропроцессора есть несколько команд, которые могут облегчить программирование алгоритмов, производящих арифметические вычисления. В них могут возникнуть различные проблемы, для разрешения которых разработчики микропроцессора предусмотрели несколько команд. Рассмотрим их в следующем разделе.

Команды преобразования типов

Что делать, если размеры operandов, участвующих в арифметических операциях, разные? Например, предположим, что в операции сложения один operand является

словом, а другой занимает двойное слово. Выше сказано, что в операции сложения должны участвовать операнды одного формата. Если числа без знака, то выход найти просто. В этом случае можно на базе исходного операнда сформировать новый (формата двойного слова), старшие разряды которого просто заполнить нулями. Сложнее ситуация для чисел со знаком: как динамически, в ходе выполнения программы, учесть знак операнда? Для решения подобных проблем в системе команд микропроцессора есть так называемые *команды преобразования типа*. Эти команды расширяют байты в слова, слова — в двойные слова и двойные слова — в четырехбайтные слова (64-разрядные значения). Команды преобразования типа особенно полезны при преобразовании целых со знаком, так как они автоматически заполняют старшие биты вновь формируемого операнда значениями знакового бита старого объекта. Эта операция приводит к целым значениям того же знака и той же величины, что и исходная, но уже в более длинном формате. Подобное преобразование называется *операцией распространения знака*.

Существуют два вида команд преобразования типа:

1. Команды без operandов — эти команды работают с фиксированными регистрами:
 - **cbw** (Convert Byte to Word) — команда преобразования байта (в регистре **al**) в слово (в регистре **ax**) путем распространения значения старшего бита **al** на все биты регистра **ah**;
 - **cwd** (Convert Word to Double) — команда преобразования слова (в регистре **ax**) в двойное слово (в регистрах **dx:ax**) путем распространения значения старшего бита **ax** на все биты регистра **dx**;
 - **cwd** (Convert Word to Double) — команда преобразования слова (в регистре **ax**) в двойное слово (в регистре **eax**) путем распространения значения старшего бита **ax** на все биты старшей половины регистра **eax**;
 - **cdq** (Convert Double Word to Quarter Word) — команда преобразования двойного слова (в регистре **eax**) в четырехбайтное слово (в регистрах **edx:eax**) путем распространения значения старшего бита **eax** на все биты регистра **edx**;
2. Команды **movsx** и **movzx**, относящиеся к командам обработки строк (см. урок 11). Эти команды обладают полезным свойством в контексте нашей проблемы:
 - **movsx** *операнд_1, операнд_2* — переслать с распространением знака. Расширяет 8- или 16-разрядное значение *операнд_2*, которое может быть регистром или операндом в памяти, до 16- или 32-разрядного значения в одном из регистров, используя значение знакового бита для заполнения старших позиций *операнд_1*. Данную команду удобно использовать для подготовки operandов со знаками к выполнению арифметических действий;
 - **movzx** *операнд_1, операнд_2* — переслать с расширением нулем. Расширяет 8- или 16-разрядное значение *операнд_2* до 16- или 32-разрядного с очисткой (заполнением) нулями старших позиций *операнд_2*. Данную команду удобно использовать для подготовки operandов без знака к выполнению арифметических действий.

К примеру, вычислим значение $y = (a + b)/c$, где a, b, c — байтовые знаковые переменные (листинг 8.7).

Листинг 8.7. Вычисление простого выражения

```
<1> :prg_8_9.asm
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> a db 5
<7> b db 10
<8> c db 2
<9> y dw 0
<10> .code
<11> main: ;точка входа в программу
<12> ...
<13> xor ax,ax
<14> mov al,a
<15> cbw
<16> movsx bx,b
<17> add ax,bx
<18> idiv c ;в al – частное, в ah – остаток
<19> exit:
<20> mov ax,4c00h ;стандартный выход
<21> int 21h
<22> end main ;конец программы
```

В этой программе делимое для команды `idiv` (строка 18) готовится заранее. Так как делитель имеет размер байта, то делимое должно быть словом. С учетом этого сложение осуществляется параллельно с преобразованием размера результата в слово (строки 14–17). Например, расширение операндов со знаком производится двумя разными командами — `cbw` и `movsx`.

Другие полезные команды

В системе команд микропроцессора есть две команды — `xadd` и `neg`, которые могут быть полезны, в частности, для программирования вычислительных действий.

`xadd назначение, источник` — обмен местами и сложение. Команда позволяет выполнить последовательно два действия:

- обменять значения *назначение* и *источник*,
 - поместить на место операнда *назначение* сумму: *назначение* = *назначение* + *источник*.
- `neg операнд` — отрицание с дополнением до двух. Команда выполняет инвертирование значения *операнд*. Физически команда выполняет одно действие: *операнд* = 0 - *операнд*, то есть вычитает *операнд* из нуля.

Команду `neg операнд` можно применять для:

- смены знака;

- выполнения вычитания из константы. Дело в том, что команды `sub` и `sbb` не позволяют вычесть что-либо из константы, так как константа не может служить операндом-приемником в этих операциях. Поэтому данную операцию можно выполнить с помощью двух команд:

```
neg    ax    ;смена знака (ax)
```

```
...
```

```
add    ax,340 :фактически вычитание: (ax)=340-(ax)
```

Арифметические операции над двоично-десятичными числами

Определение и формат BCD-чисел были рассмотрены в начале этого урока. У вас справедливо может возникнуть вопрос: а зачем нужны BCD-числа? Ответ может быть следующим: BCD-числа нужны в деловых приложениях, то есть там, где числа должны быть большими и точными. Как мы уже убедились на примере двоичных чисел, операции с такими числами довольно проблематичны для языка ассемблера. К недостаткам использования двоичных чисел можно отнести следующие:

- значения величин в формате слова и двойного слова имеют ограниченный диапазон. Если программа предназначена для работы в области финансов, то ограничение суммы в рублях величиной 65 536 (для слова) или даже 4 294 967 296 (для двойного слова) будет существенно сужать сферу ее применения (да еще в наших экономических условиях — тут уж никакая деноминация не поможет);
- наличие ошибок округления. Представляете себе программу, работающую где-нибудь в банке, которая не учитывает величину остатка при действиях с целыми двоичными числами и оперирует при этом миллиардами. Не хотелось бы быть автором такой программы. Применение чисел с плавающей точкой не спасет — там существует та же проблема округления;
- представление большого объема результатов в символьном виде (ASCII-коде). Деловые программы не просто выполняют вычисления; одной из целей их использования является оперативная выдача информации пользователю. Для этого, естественно, информация должна быть представлена в символьном виде. Перевод чисел из двоичного кода в ASCII-код, как мы уже видели, требует определенных вычислительных затрат. Число с плавающей точкой еще труднее перевести в символьный вид. А вот если посмотреть на шестнадцатеричное представление неупакованной десятичной цифры (в начале нашего урока) и на соответствующий ей символ в таблице ASCII, то видно, что они отличаются на величину 30h. Таким образом, преобразование в символьный вид и обратно получается намного проще и быстрее.

Наверняка вы уже убедились в важности овладения хотя бы основами действий с десятичными числами. Далее рассмотрим особенности выполнения основных арифметических операций с десятичными числами. Для предупреждения возможных вопросов отметим сразу тот факт, что отдельных команд сложения, вычита-

ния, умножения и деления BCD-чисел нет. Сделано это по вполне понятным причинам: размерность таких чисел может быть сколь угодно большой. Складывать и вычитать можно двоично-десятичные числа как в упакованном формате, так и в неупакованном, а вот делить и умножать можно только неупакованные BCD-числа. Почему это так, будет видно из дальнейшего обсуждения.

Неупакованные BCD-числа

Сложение

Рассмотрим два случая сложения.

Пример 8.9. Результат сложения не больше 9

$$\begin{array}{r} 6 = 0000 \ 0110 \\ + \\ 3 = 0000 \ 0011 \\ = \\ 9 = 0000 \ 1001 \end{array}$$

Переноса из младшей тетрады в старшую нет. Результат правильный.

Пример 8.10. Результат сложения больше 9

$$\begin{array}{r} 06 = 0000 \ 0110 \\ + \\ 07 = 0000 \ 0111 \\ = \\ 13 = 0000 \ 1101 \end{array}$$

То есть мы получили уже не BCD-число. Результат неправильный. Правильный результат в неупакованном BCD-формате должен быть таким: 0000 0001 0000 0011 в двоичном представлении (или 13 в десятичном). Проанализировав данную проблему при сложении BCD-чисел (и подобные проблемы при выполнении других арифметических действий) и возможные пути ее решения, разработчики системы команд микропроцессора решили не вводить специальные команды для работы с BCD-числами, а ввести несколько корректировочных команд. Назначение этих команд — в корректировке результата работы обычных арифметических команд для случаев, когда операнды в них являются BCD-числами. В случае сложения в примере 8.10 видно, что полученный результат нужно корректировать. Для коррекции операции сложения двух однозначных неупакованных BCD-чисел в системе команд микропроцессора существует специальная команда

aaa (ASCII Adjust for Addition) — коррекция результата сложения для представления в символьном виде.

Эта команда не имеет operandов. Она работает неявно только с регистром a1 и анализирует значение его младшей тетрады. Если это значение меньше 9, то флаг cf сбрасывается в 0, и осуществляется переход к следующей команде. Если это значение больше 9, то выполняются следующие действия:

○ к содержимому младшей тетрады *a1* (но не к содержимому всего регистра!) прибавляется 6, тем самым значение десятичного результата корректируется в правильную сторону;

○ флаг *cf* устанавливается в 1, тем самым фиксируется перенос в старший разряд для того, чтобы его можно было учесть в последующих действиях.

Так, в примере 8.10, предполагая, что значение суммы 0000 1101 находится в *a1*, после команды *aaa* в регистре будет $1101 + 0110 = 0011$, то есть двоичное 0000 0011 или десятичное 3, а флаг *cf* установится в 1, то есть перенос запомнился в микропроцессоре. Далее программисту нужно будет использовать команду сложения *adc*, которая учетет перенос из предыдущего разряда. Приведем пример программы сложения двух неупакованных BCD-чисел.

Листинг 8.8. Сложение неупакованных BCD-чисел

```
<1>          ;prg_8_8.asm
<2>          ...
<3>          .data
<4>          len    equ    2      ;разрядность числа
<5>          b      db     1,7    ;неупакованное число 71
<6>          c      db     4,5    ;неупакованное число 54
<7>          sum   db     3 dup (0)
<8>          .code
<9>          main:           ;точка входа в программу
<10>         ...
<11>         xor    bx,bx
<12>         mov    cx,len
<13>         m1:
<14>         mov    a1,b[bx]
<15>         adc    a1,c[bx]
<16>         aaa
<17>         mov    sum[bx],a1
<18>         inc    bx
<19>         loop   m1
<20>         adc    sum[bx],0
<21>         ...
          exit:
```

В листинге 8.8 есть несколько интересных моментов, над которыми есть смысл поразмыслить. Начнем с описания BCD-чисел. Из строк 5 и 6 видно, что порядок их ввода обратен нормальному, то есть цифры младших разрядов расположены по меньшему адресу. Но это вполне логично по нескольким причинам: во-первых, такой порядок удовлетворяет общему принципу представления данных для микропроцессоров Intel, во-вторых, это очень удобно для поразрядной обработки неупакованных BCD-чисел, так как каждое из них занимает один байт. Хотя, повторюсь, программист сам волен выбирать способ описания BCD-чисел в сегменте данных. Строки 14–15 содержат команды, которые складывают цифры в очередных разрядах BCD-чисел, при этом учитывается возможный перенос из младшего разряда. Команда *aaa* в строке 16 корректирует результат сложения, формируя в *a1* BCD-цифру и, при необходимости, устанавливая в 1 флаг *cf*. Строки

ка 20 учитывает возможность переноса при сложении цифр из самых старших разрядов чисел. Результат сложения формируется в поле `sum`, описанном в строке 7.

Вычитание

Ситуация здесь вполне аналогична сложению. Рассмотрим те же случаи.

Пример 8.11. Результат вычитания не больше 9

$$6 = 0000\ 0110$$

—

$$3 = 0000\ 0011$$

=

$$3 = 0000\ 0011$$

Как видим, заема из старшей тетрады нет. Результат верный и корректировки не требует.

Пример 8.12. Результат вычитания больше 9

$$6 = 0000\ 0110$$

—

$$7 = 0000\ 0111$$

=

$$-1 = 1111\ 1111$$

Вычитание проводится по правилам двоичной арифметики. Поэтому результат не является BCD-числом. Правильный результат в неупакованном BCD-формате должен быть 9 (0000 1001 в двоичной системе счисления). При этом предполагается заем из старшего разряда, как при обычной команде вычитания, то есть в случае с BCD-числами фактически должно быть выполнено вычитание 16 – 7. Таким образом, видно, что, как и в случае сложения, результат вычитания нужно корректировать. Для этого существует специальная команда

`aas` (ASCII Adjust for Subtraction) – коррекция результата вычитания для представления в символьном виде.

Команда `aas` также не имеет операндов и работает с регистром `a1`, анализируя его младшую тетраду следующим образом: если ее значение меньше 9, то флаг `sf` сбрасывается в 0, и управление передается следующей команде. Если значение тетрады в `a1` больше 9, то команда `aas` выполняет следующие действия:

- из содержимого младшей тетрады регистра `a1` (заметьте – не из содержимого всего регистра) вычитает 6;
- обнуляет старшую тетраду регистра `a1`;
- устанавливает флаг `sf` в 1, тем самым фиксируя воображаемый заем из старшего разряда.

Понятно, что команда `aas` применяется вместе с основными командами вычитания `sub` и `sbb`. При этом команду `sub` есть смысл использовать только один раз при вычитании самых младших цифр operandов, далее должна применяться команда

sbb, которая будет учитывать возможный заем из старшего разряда. В листинге 8.9 мы обходимся одной командой **sbb**, которая в цикле производит поразрядное вычитание двух BCD-чисел.

Листинг 8.9. Вычитание неупакованных BCD-чисел

```
<1> :prg_8_9.asm
<2> masm
<3> model small
<4> stack 256
<5> .data           ;сегмент данных
<6> b    db   1,7  ;неупакованное число 71
<7> c    db   4,5  ;неупакованное число 54
<8> subs db   2 dup (0)
<9> .code
<10> main:          ;точка входа в программу
<11>     mov  ax,@data  ;связываем регистр dx с сегментом
<12>     mov  ds,ax  ;данных через регистр ax
<13>     xor  ax,ax  ;очищаем ax
<14>     len  equ  2   ;разрядность чисел
<15>     xor  bx,bx
<16>     mov  cx,len  ;загрузка в cx счетчика цикла
<17> m1:
<18>     mov  al,b[bx]
<19>     sbb  al,c[bx]
<20>     aas
<21>     mov  subs[bx],al
<22>     inc  bx
<23>     loop m1
<24>     jc   m2      ;анализ флага заема .
<25>     jmp  exit
<26> m2: ...
<27> exit:
<28>     mov  ax,4c00h  ;стандартный выход
<29>     int  21h
<30> end  main        ;конец программы
```

Данная программа не требует особых пояснений, когда уменьшаемое больше вычитаемого. Поэтому обратите внимание на строку 24. С ее помощью мы предусматриваем случай, когда после вычитания старших цифр чисел был зафиксирован факт заема. Это говорит о том, что вычитаемое было больше уменьшаемого, в результате чего разность будет неправильной. Эту ситуацию нужно как-то обработать. С этой целью в строке 24 командой **jc** анализируется флаг **sf**. По результату этого анализа мы уходим на ветку программы, обозначенную меткой **m2**, где и будут выполняться некоторые действия. Набор этих действий сильно зависит от конкретного алгоритма обработки, поэтому поясним только суть действий, которые может выполнять соответствующий фрагмент программы. Для этого посмотрим в отладчике, как наша программа выполнит вычитание $50 - 74$ (правильный ответ -24). То, что вы увидите в окне **Dump** отладчика, в поле, соответствующем адресу **subs**, будет далеко от истинного ответа. Что делает в этом случае человек? Он просто выполняет вычитание $74 - 50 = 24$ и рассматривает результат как имеющий знак

минус. Так как у микропроцессора нет средств обработки подобной ситуации, то фрагмент программы, обозначенный меткой `m2`, может поменять уменьшаемое и вычитаемое местами, выполнить вычитание и где-то отметить тот факт, что разность, на самом деле, нужно рассматривать как отрицательное число. Но ключевой момент здесь все-таки тот, что микропроцессор с помощью флага `sf` сигнализирует нам об этой особой ситуации.

Умножение

На примере сложения и вычитания неупакованных чисел стало понятно, что стандартных алгоритмов для выполнения этих действий над BCD-числами нет, и программист должен сам, исходя из требований к своей программе, реализовать эти операции. Реализация двух оставшихся операций — умножения и деления — еще более сложна. В системе команд микропроцессора присутствуют только средства для производства умножения и деления одноразрядных неупакованных BCD-чисел.

Для того чтобы умножать числа произвольной размерности, нужно реализовать процесс умножения самостоятельно, взяв за основу некоторый алгоритм умножения, например «в столбик». Позже мы рассмотрим пример программы, выполняющей умножение десятичных чисел произвольной размерности.

Для того чтобы перемножить два одноразрядных BCD-числа, необходимо:

- поместить один из сомножителей в регистр `a1` (как того требует команда `mul`);
- поместить второй операнд в регистр или память, отведя байт;
- перемножить сомножители командой `mul` (результат, как и положено, будет в `ax`);
- результат, конечно, получится в двоичном коде, поэтому его нужно скорректировать.

Для коррекции результата после умножения применяется специальная команда

`aam` (ASCII Adjust for Multiplication) — коррекция результата умножения для представления в символьном виде.

Она не имеет operandов и работает с регистром `ax` следующим образом:

- делит `a1` на 10;
- результат деления записывается так: частное — в `a1`, остаток — в `ah`.

В результате после выполнения команды `aam` в регистрах `a1` и `ah` находятся правильные двоично-десятичные цифры произведения двух цифр.

В листинге 8.10 приведен пример умножения BCD-числа произвольной размерности на однозначное BCD-число.

Листинг 8.10. Умножение неупакованных BCD-чисел

```
<1>  masm
<2>  model small
<3>  stack 256
<4>  .data
```

```

<5> b      db    6,7    ;неупакованное число 76
<6> c      db    4      ;неупакованное число 4
<7> proizvdb 4 dup (0)
<8> .code
<9> main:          ;точка входа в программу
<10>     mov  ax,@data
<11>     mov  ds,ax
<12>     xor  ax,ax
<13> len   equ   2      ;размерность сомножителя 1
<14>     xor  bx,bx
<15>     xor  si,si
<16>     xor  di,di
<17>     mov  cx,len   ;в cx длина наибольшего сомножителя 1
<18> m1:
<19>     mov  al,b[si]
<20>     mul  c
<21>     aam
<22>     adc  al,d1   ;коррекция умножения
<23>     aaa
<24>     mov  d1,ah   ;учли предыдущий перенос
<25>     mov  proizv[bx],al ;скорректировали результат сложения с переносом
<26>     inc  si
<27>     inc  bx
<28>     loop m1      ;цикл на метку m1
<29>     mov  proizv[bx],d1 ;запомнили перенос
<30> exit:
<31>     mov  ax,4c00h
<32>     int  21h
<33> end  main

```

Данную программу можно легко модифицировать для умножения BCD-чисел произвольной длины. Для этого достаточно представить алгоритм умножения в «столбик». Листинг 8.10 можно использовать для получения частичных произведений в этом алгоритме. После их сложения со сдвигом получится искомый результат. Попробуйте выполнить разработку этой программы самостоятельно.

Перед окончанием обсуждения команды `aam` необходимо отметить еще один вариант ее применения. Эту команду можно применять для преобразования двоичного числа в регистре `al` в неупакованное BCD-число, которое будет размещено в регистре `ax`: старшая цифра результата — в `ah`, младшая — в `al`. Понятно, что двоичное число должно быть в диапазоне 0...99.

Деление

Процесс выполнения операции деления двух неупакованных BCD-чисел несколько отличается от других, рассмотренных ранее, операций с ними. Здесь также требуются действия по коррекции, но они должны выполняться до основной операции, выполняющей непосредственно деление одного BCD-числа на другое BCD-число. Предварительно в регистре `ax` нужно получить две неупакованные BCD-цифры делимого. Это делает программист удобным для него способом. Далее нужно выдать команду `aad`:

aad (ASCII Adjust for Division) — коррекция деления для представления в символьном виде.

Команда не имеет operandов и преобразует двузначное неупакованное BCD-число в регистре **ах** в двоичное число. Это двоичное число впоследствии будет играть роль делимого в операции деления. Кроме преобразования, команда **aad** помещает полученное двоичное число в регистр **a1**. Делимое, естественно, будет двоичным числом из диапазона 0...99. Алгоритм, по которому команда **aad** осуществляется это преобразование, состоит в следующем:

- умножить старшую цифру исходного BCD-числа в **ах** (содержимое **ah**) на 10;
- выполнить сложение **ah + a1**, результат которого (двоичное число) занести в **a1**;
- обнулить содержимое **ah**.

Далее программисту нужно выдать обычную команду деления **div** для выполнения деления содержимого **ах** на одну BCD-цифру, находящуюся в байтовом регистре или байтовой ячейке памяти. Деление неупакованных BCD-чисел иллюстрируется листингом 8.11.

Листинг 8.11. Деление неупакованных BCD-чисел

```
<1> :prg_8_11.asm
<2> ...
<3> .data :сегмент данных
<4> b db 1,7 ;неупакованное BCD-число 71
<5> c db 4 ;
<6> ch db 2 dup (0)
<7> .code :сегмент кода
<8> main: ;точка входа в программу
<9> ...
<10>    mov a1,b
<11>    aad ;коррекция перед делением
<12>    div c ;в a1 BCD – частное, в ah BCD – остаток
<13> ...
<14>    exit:
```

Аналогично **aam**, команде **aad** можно найти и другое применение — использовать ее для перевода неупакованных BCD-чисел из диапазона 0...99 в их двоичный эквивалент.

Для деления чисел большей разрядности, так же, как и в случае умножения, нужно реализовывать свой алгоритм, например в «столбик», либо найти более оптимальный путь. Любопытный и настойчивый читатель, возможно, самостоятельно разработает эти программы. Но это делать совсем необязательно. На диске, прилагаемой к книге, в каталоге данного урока приведены тексты макрорукоманд, которые выполняют четыре основных арифметических действия с BCD-числами любой разрядности.

Упакованные BCD-числа

Как уже отмечалось выше, упакованные BCD-числа можно только складывать и вычитать. Для выполнения других действий над ними их нужно дополнитель-

преобразовывать либо в неупакованный формат, либо в двоичное представление. Из-за того что упакованные BCD-числа представляют не слишком большой интерес, мы их рассмотрим кратко.

Сложение

Вначале разберемся с сутью проблемы и попытаемся сложить два двузначных упакованных BCD-числа.

Пример 8.13. Сложение упакованных BCD-чисел

$$\begin{array}{r} 67 = 0110 \ 0111 \\ + \\ 75 = 0111 \ 0101 \\ = \\ 142 = 1101 \ 1100 = 220 \end{array}$$

Как видим, в двоичном виде результат равен 1101 1100 (или 220 в десятичном представлении), что неверно. Это происходит по той причине, что микропроцессор не подозревает о существовании BCD-чисел и складывает их по правилам сложения двоичных чисел. На самом деле результат в двоично-десятичном виде должен быть равен 0001 0100 0010 (или 142 в десятичном представлении). Читатель видит, что, как и для неупакованных BCD-чисел, для упакованных BCD-чисел существует необходимость как-то корректировать результаты арифметических операций. Микропроцессор предоставляет для этого команду `daa`:

`daa` (Decimal Adjust for Addition) – коррекция результата сложения для представления в десятичном виде.

Команда `daa` преобразует содержимое регистра `a1` в две упакованные десятичные цифры по алгоритму, приведенному в Справочнике.

Получившаяся в результате сложения единица (если результат сложения больше 99) запоминается в флаге `cf`, тем самым учитывается перенос в старший разряд.

Проиллюстрируем сказанное на примере сложения двух двузначных BCD-чисел в упакованном формате (листинг 8.12).

Листинг 8.12. Сложение упакованных BCD-чисел

```
<1>    :prg_8_12.asm
<2>    ...
<3>    .data          ;сегмент данных
<4>    b    db    17h   ;упакованное число 17
<5>    c    db    45h   ;упакованное число 45
<6>    sum  db    2 dup (0)
<7>    .code          ;сегмент кода
<8>    main:           ;точка входа в программу
<9>    ...
<10>       mov  a1,b
<11>       add  a1,c
<12>       daa
```

```

<13>      jnc    $+6    ;переход через команду, если результат <= 99
<14>      mov    sum+1,ah    ;учет переноса при сложении (результат > 99)
<15>      mov    sum,al    ;младшие упакованные цифры результата
<16>  exit:

```

В приведенном примере все достаточно прозрачно; единственное, на что следует обратить внимание – это описание упакованных BCD-чисел и порядок формирования результата. Результат формируется в соответствии с основным принципом работы микропроцессоров Intel: младший байт по младшему адресу.

Вычитание

Аналогично сложению, микропроцессор рассматривает упакованные BCD-числа как двоичные, и, соответственно, выполняет вычитание BCD-чисел как двоичных (см. пример 8.14). Выполним вычитание 67–75. Так как микропроцессор выполняет вычитание способом сложения, то и мы последуем этому:

Пример 8.14. Вычитание упакованных BCD-чисел

$$\begin{array}{r}
 67 = 0110\ 0111 \\
 + \\
 -75 = 1011\ 0101 \\
 = \\
 -8 = 0001\ 1100 = 28 \ ?
 \end{array}$$

Как видим, результат равен 28 в десятичной системе счисления, что является абсурдом. В двоично-десятичном коде результат должен быть равен 0000 1000 (или 8 в десятичной системе счисления). При программировании вычитания упакованных BCD-чисел программист, как и при вычитании неупакованных BCD-чисел, должен сам осуществлять контроль за знаком. Это делается с помощью флага cf, который фиксирует заем из старших разрядов. Само вычитание BCD-чисел осуществляется простыми командами вычитания sub или sbb. Коррекция результата осуществляется командой das:

das (Decimal Adjust for Subtraction) – коррекция результата вычитания для представления в десятичном виде.

В Справочнике описан алгоритм, по которому команда das преобразует содержимое регистра al в две упакованные десятичные цифры.

Подведем некоторые итоги:

- Микропроцессор имеет достаточно мощные средства для реализации вычислительных операций. Для этого у него есть блок целочисленных операций и блок операций с плавающей точкой. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.
- Команды целочисленных операций работают с данными двух типов: двоичными и двоично-десятичными числами (BCD-числами).

- Двоичные данные могут либо иметь знак, либо не иметь такового. Микропроцессор, на самом деле, не различает числа со знаком и без. Он помогает лишь отслеживать изменение состояния некоторых битов операндов и состояние отдельных флагов. Операции сложения и вычитания чисел со знаком и без знака проводятся одним устройством и по единым правилам.
- Контроль за правильностью результатов и их надлежащей интерпретацией полностью лежит на программисте. Он должен контролировать состояние флагов cf и of регистра eflags во время вычислительного процесса.
- Для операций с числами без знака нужно контролировать флаг cf. Установка его в 1 сигнализирует о том, что число вышло за разрядную сетку операндов.
- Для чисел со знаком установка флага of в 1 говорит о том, что в результате сложения чисел одного знака результат выходит за границу допустимых значений чисел со знаком в данном формате, и сам результат меняет знак (пропадает порядок).
- По результатам выполнения арифметических операций устанавливаются также флаги pf, zf и sf.
- В отличие от команд сложения и вычитания, команды умножения и деления позволяют учитывать знаки операндов.
- Арифметические команды очень «капризны» к размерности операндов, поэтому в систему команд микропроцессора включены специальные команды, позволяющие отслеживать эту характеристику.
- Двоичные данные имеют довольно большой, но ограниченный диапазон значений. Для коммерческих приложений этот диапазон слишком мал, поэтому в архитектуру микропроцессора введены средства для работы с так называемыми двоично-десятичными (BCD) данными.
- Двоично-десятичные данные представляются в двух форматах, упакованном и неупакованном. Наиболее универсальным является неупакованный формат.

9

УРОК

Логические команды

-
- Краткое описание группы логических команд
 - Команды для выполнения логических операций
 - Организация работы с отдельными битами
 - Сдвиги
-

Наряду со средствами арифметических вычислений, система команд микропроцессора имеет также средства логического преобразования данных. Под *логическими* понимаются такие преобразования данных, в основе которых лежат правила *формальной логики*. Формальная логика работает на уровне утверждений *истинно* и *ложно*. Для микропроцессора это, как правило, означает 1 и 0, соответственно. Для компьютера язык нулей и единиц является родным, но минимальной единицей данных, с которой работают машинные команды, является байт. Однако на системном уровне часто необходимо иметь возможность работать на предельно низком уровне — на уровне бит.

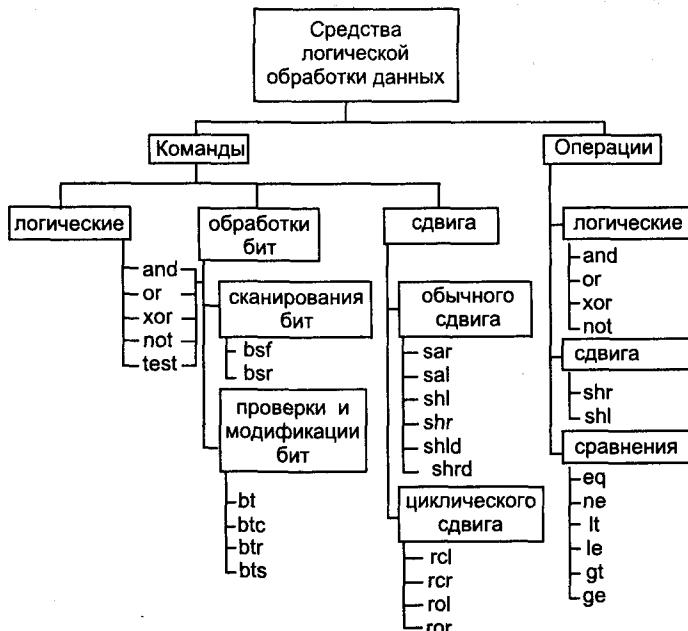


Рис. 9.1. Средства микропроцессора для работы с логическими данными

К средствам логического преобразования данных относятся *логические команды* и *логические операции*. На рис. 9.1 показаны средства микропроцессора для организации работы с данными по правилам формальной логики. Они разбиты на две группы: команды и операции. Команды рассматриваются на этом уроке. Операции были изучены нами на уроке 5. Напомню, что операнд команды ассемблера в общем случае может представлять собой выражение, которое, в свою

очередь, является комбинацией операторов и операндов. Среди этих операторов могут быть и операторы, реализующие логические операции над объектами выражения.

Перед знакомством с этими средствами давайте посмотрим, что же представляют собой сами логические данные, и какие операции над ними производятся.

Логические данные

Теоретической базой для логической обработки данных является формальная логика. Существует несколько систем логики. Одна из наиболее известных — это *исчисление высказываний*. *Высказывание* — это любое утверждение, о котором можно сказать, что оно либо *истинно*, либо *ложно*. *Исчисление высказываний* представляет собой совокупность правил, используемых для определения истинности или ложности некоторой комбинации высказываний.

Исчисление высказываний очень гармонично сочетается с принципами работы компьютера и основными методами его программирования. Все аппаратные компоненты компьютера построены на логических микросхемах. Система представления информации в компьютере на самом нижнем уровне основана на понятии *бит*. Бит, имея всего два состояния — 0 (ложно) и 1 (истинно), естественным образом вписывается в исчисление высказываний.

Согласно теории, над высказываниями (над битами) могут выполняться следующие *логические операции*:

- *отрицание* (логическое *НЕ*) — логическая операция над одним операндом, результатом которой является величина, обратная значению исходного операнда. Эта операция однозначно характеризуется следующей *таблицей истинности*¹ (табл. 9.1);

Таблица 9.1. Таблица истинности для логического отрицания

| | | |
|--------------------|---|---|
| Значение операнда | 0 | 1 |
| Результат операции | 1 | 0 |

- *логическое сложение* (логическое *включающее ИЛИ*) — логическая операция над двумя операндами, результатом которой является «истина» (1), если один или оба операнда имеют значение «истина» (1), и «ложь» (0), если оба операнда имеют значение «ложь» (0). Эта операция описывается с помощью следующей таблицы истинности (табл. 9.2);

Таблица 9.2. Таблица истинности для логического включающего ИЛИ

| | | | | |
|---------------------|---|---|---|---|
| Значение операнда 1 | 0 | 0 | 1 | 1 |
| Значение операнда 2 | 0 | 1 | 0 | 1 |
| Результат операции | 0 | 1 | 1 | 1 |

¹ Таблица истинности — таблица результатов логических операций в зависимости от значений исходных операндов.

- **логическое умножение** (логическое *И*) – логическая операция над двумя операндами, результатом которой является «истина» (1) только в том случае, если оба операнда имеют значение «истина» (1). Во всех остальных случаях значение операции – «ложь» (0). Эта операция описывается с помощью следующей таблицы истинности (табл. 9.3);

Таблица 9.3. Таблица истинности для логического И

| | | | | |
|---------------------|---|---|---|---|
| Значение операнда 1 | 0 | 0 | 1 | 1 |
| Значение операнда 2 | 0 | 1 | 0 | 1 |
| Результат операции | 0 | 0 | 0 | 1 |

- **логическое исключающее сложение** (логическое исключающее *ИЛИ*) – логическая операция над двумя операндами, результатом которой является «истина» (1), если только один из двух операндов имеет значение «истина» (1), и ложь (0), если оба операнда имеют значения «ложь» (0) или «истина» (1). Эта операция описывается с помощью следующей таблицы истинности (табл. 9.4);

Таблица 9.4. Таблица истинности для логического исключающего ИЛИ

| | | | | |
|---------------------|---|---|---|---|
| Значение операнда 1 | 0 | 0 | 1 | 1 |
| Значение операнда 2 | 0 | 1 | 0 | 1 |
| Результат операции | 0 | 1 | 1 | 0 |

Система команд микропроцессора содержит пять команд, поддерживающих данные операции. Эти команды выполняют логические операции над битами операндов. Размерность операндов, естественно, должна быть одинакова. Например, если размерность операндов равна слову (16 битов), то логическая операция выполняется сначала над нулевыми битами операндов, и ее результат записывается на место бита 0 результата. Далее команда последовательно повторяет эти действия над всеми битами с первого до пятнадцатого.

Возможные варианты размерности операндов для каждой команды можно найти в Справочнике.

Логические команды

В системе команд микропроцессора есть следующий набор команд, поддерживающих работу с логическими данными:

and *операнд_1, операнд_2* – операция логического умножения. Команда выполняет поразрядно логическую операцию *И* (*конъюнкцию*) над битами операндов *операнд_1* и *операнд_2*. Результат записывается на место *операнд_1*.

or *операнд_1, операнд_2* – операция логического сложения. Команда выполняет поразрядно логическую операцию *ИЛИ* (*дизъюнкцию*) над битами операндов *операнд_1* и *операнд_2*. Результат записывается на место *операнд_1*.

хор операнд_1, операнд_2 — операция логического исключающего сложения. Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами operandов *операнд_1* и *операнд_2*. Результат записывается на место *операнд_1*.

test операнд_1, операнд_2 — операция «проверить» (способом логического умножения). Команда выполняет поразрядно логическую операцию И над битами operandов *операнд_1* и *операнд_2*. Состояние operandов остается прежним, изменяются только флаги zf, sf, и rf, что дает возможность анализировать состояние отдельных битов operandана без изменения их состояния.

not операнд — операция логического отрицания. Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита operandана. Результат записывается на место operandана.

Для представления роли логических команд в системе команд микропроцессора очень важно понять области их применения и типовые приемы их использования при программировании. Далее мы будем рассматривать логические команды в контексте обработки последовательности битов.

Очень часто некоторая ячейка памяти должна играть роль индикатора, показывая, например, занятость некоторого программного или аппаратного ресурса. Так как эта ячейка может принимать только два значения — занято (1) или свободно (0), то отводить под нее целый байт очень расточительно, логичнее для этой цели использовать бит. А если таких индикаторов много? Если их объединить в пределах одного байта или слова, то может получиться довольно существенная экономия памяти. Посмотрим, что могут сделать для этого логические команды. Для лучшего усвоения данного материала представьте, что вам поручили запрограммировать работу елочной гирлянды. Очень красиво смотрятся гирлянды, управляемые прерывателями, особенно если они работают по некоторому алгоритму. Вы можете представить себе прерыватель для гирлянды из 8, 16 или 32 лампочек. Естественным является подход, при котором каждой лампочке соответствует определенный бит 8-, 16- или 32-битного поля. Для логической завершенности постановки данной задачи можно предложить следующий вариант управления: вход прерывателя подключен к одному из доступных для подключения внешних портов ввода-вывода компьютера, на который и будет выдаваться управляющая битовая последовательность.

С помощью логических команд возможно выделение отдельных битов в operandе с целью их установки, сброса, инвертирования или просто проверки на определенное значение. Для организации подобной работы с битами *операнд_2* обычно играет роль *маски*. С помощью установленных в 1 битов этой маски и определяются нужные для конкретной операции биты *операнд_1*. Покажем, какие логические команды могут применяться для этой цели.

Для установки определенных разрядов (бит) в 1 применяется команда
or операнд_1, операнд_2

В этой команде *операнд_2*, выполняющий роль маски, должен содержать единичные биты на месте тех разрядов, которые должны быть установлены в 1 в *операнд_1*.

or eax,10b :установить 1-й бит в регистре eax

Для сброса определенных разрядов (битов) в 0 применяется команда

and *операнд_1, операнд_2*

В этой команде *операнд_2*, выполняющий роль маски, должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в *операнд_1*

and eax,0xffffffffdh ;сбросить в 0 1-й бит в регистре eax

Команда **xor** *операнд_1, операнд_2* применяется:

- для выяснения того, какие биты в *операнд_1* и *операнд_2* различаются;
- для инвертирования состояния заданных бит в *операнд_1*.

Интересующие нас биты маски (*операнд_2*) при выполнении команды **xor** должны быть единичными, остальные — нулевыми.

**xoreax,10b ;инвертировать 1-й бит в регистре eax
jz mes ;переход, если 1-й бит в a1 был единичным**

Для проверки состояния заданных бит применяется команда

test *операнд_1, операнд_2* (проверить *операнд_1*)

Проверяемые биты *операнд_1* в маске (*операнд_2*) должны иметь единичное значение. Алгоритм работы команды **test** подобен алгоритму команды **and**, но он не меняет значения *операнд_1*. Результатом команды является установка значения флага нуля *zf*:

- если *zf* = 0, то в результате логического умножения получился нулевой результат, то есть один единичный бит маски не совпал с соответствующим единичным битом *операнд_1*;
- если *zf* = 1, то в результате логического умножения получился ненулевой результат, то есть хотя бы один единичный бит маски совпал с соответствующим единичным битом *операнд_1*.

**test eax,00000010h
jz m1 ;переход если 4-й бит равен 1**

Как видно из примера, для реакции на результат команды **test** целесообразно использовать команду перехода **jnz** метка (**Jump if Not Zero**) — переход, если флаг нуля *zf* ненулевой, или команду с обратным действием — **jz** метка (**Jump if Zero**) — переход, если флаг нуля *zf* = 0.

Следующие две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала, так и от конца операнда:

bsf *операнд_1, операнд_2* (Bit Scanning Forward) — сканирование бит вперед. Команда просматривает (сканирует) биты *операнд_2* от младшего к старшему (от бита 0 до старшего бита) в поисках первого бита, установленного в 1. Если такой обнаруживается, в *операнд_1* заносится номер этого бита в виде целочисленного значения. Если все биты *операнд_2* равны 0, то флаг нуля *zf* устанавливается в 1, в противном случае флаг *zf* сбрасывается в 0.

```

    mov  al,02h
    bsf  bx,al ;bx=1
    jz   m1 ;переход, если al=00h
    ...

```

bsr *операнд_1, операнд_2* (Bit Scaning Reset) – сканирование битов в обратном порядке. Команда просматривает (сканирует) биты *операнд_2* от старшего к младшему (от старшего бита к биту 0) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в *операнд_1* заносится номер этого бита в виде целочисленного значения. При этом важно, что позиция первого единичного бита слева отсчитывается все равно относительно бита 0. Если все биты *операнд_2* равны 0, то флаг нуля zf устанавливается в 1, в противном случае флаг zf сбрасывается в 0.

Листинг 9.1 демонстрирует пример применения команд **bsr** и **bsf**. Введите код и исследуйте работу программы в отладчике (в частности, обратите внимание на то, как меняется содержимое регистра bx после команд **bsf** и **bsr**).

Листинг 9.1. Сканирование битов

```

;prg_9_1.asm
masm
model small
stack 256
.data           ;сегмент данных
.code           ;сегмент кода
main:          ;точка входа в программу
    mov ax,@data
    mov ds,ax
    ...
.486           ;это обязательно
    xor ax,ax
    mov al,02h
    bsf bx,ax      ;bx=1
    jz m1          ;переход, если al=00h
    bsr bx,ax
m1:
    ...
    mov ax,4c00h   ;стандартный выход
    int 21h
endmain

```

В последних моделях микропроцессоров Intel в группе логических команд появилось еще несколько команд, которые позволяют осуществить доступ к одному конкретному биту операнда. Операнд может находиться как в памяти, так и в регистре общего назначения. Положение бита задается смещением бита относительно младшего бита операнда. Значение смещения может задаваться как в виде непосредственного значения, так и содержаться в регистре общего назначения. В качестве значения смещения вы можете использовать результаты работы команд **bsr** и **bsf**. Все команды присваивают значение выбранного бита флагу cf.

bt *операнд, смещение_бита* (Bit Test) — проверка бита. Команда переносит значение бита в флаг cf.

```
bt    ax,5  ;проверить значение бита 5
```

```
jnc  m1      ;переход, если бит = 0
```

bts *операнд, смещение_бита* (Bit Test and Set) — проверка и установка бита. Команда переносит значение бита в флаг cf и затем устанавливает проверяемый бит в 1.

```
mov  ax,10
```

```
bts  pole,ax  ;проверить и установить 10-й бит в pole
```

```
jc   m1      ;переход, если проверяемый бит был равен 1
```

btr *операнд, смещение_бита* (Bit Test and Reset) — проверка и сброс бита. Команда переносит значение бита в флаг cf и затем устанавливает этот бит в 0.

btc *операнд, смещение_бита* (Bit Test and Convert) — проверка и инвертирование бита. Команда переносит значение бита в флаг cf и затем инвертирует значение этого бита.

Команды сдвига

Команды этой группы также обеспечивают манипуляции над отдельными битами операндов, но иным способом, чем логические команды, рассмотренные выше. Все команды сдвига перемещают биты в поле операнда влево или вправо, в зависимости от кода операции. Все команды сдвига имеют одинаковую структуру:

коп *операнд, счетчик_сдвигов*

Количество сдвигаемых разрядов, *счетчик_сдвигов*, располагается, как видите, на месте второго операнда и может задаваться двумя способами:

- статически; предполагает задание фиксированного значения с помощью непосредственного операнда;
- динамически; занесение значения *счетчика_сдвигов* в регистр cl перед выполнением команды сдвига.

Исходя из размерности регистра cl, понятно, что значение *счетчика_сдвигов* может лежать в диапазоне от 0 до 255. Но на самом деле это не совсем так. В целях оптимизации микропроцессор воспринимает только значения пяти младших битов счетчика, то есть значение лежит в диапазоне от 0 до 31. В последних моделях микропроцессора, в том числе и в микропроцессоре Pentium, есть дополнительные команды, позволяющие делать 64-разрядные сдвиги. Мы их рассмотрим чуть позже.

Все команды сдвига устанавливают флаг переноса cf. По мере сдвига битов за пределы операнда они сначала попадают во флаг переноса, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

- команды линейного сдвига;
- команды циклического сдвига.

Линейный сдвиг

К командам этого типа относятся команды, осуществляющие сдвиг по следующему алгоритму:

- очередной «выдвигаемый» бит устанавливает флаг cf;
- бит, вводимый в операнд с другого конца, имеет значение 0;
- при сдвиге очередного бита он переходит во флаг cf, при этом значение предыдущего сдвинутого бита теряется!

Команды линейного сдвига делятся на два подтипа:

- команды логического линейного сдвига;
- команды арифметического линейного сдвига.

К командам логического линейного сдвига относятся следующие:

shl *операнд.счетчик_сдвигов* (Shift Logical Left) — логический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением *счетчик_сдвигов*. Справа (в позицию младшего бита) вписываются нули;

shr *операнд.счетчик_сдвигов* (Shift Logical Right) — логический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением *счетчик_сдвигов*. Слева (в позицию старшего, знакового бита) вписываются нули.

На рис. 9.2 показан принцип работы этих команд.

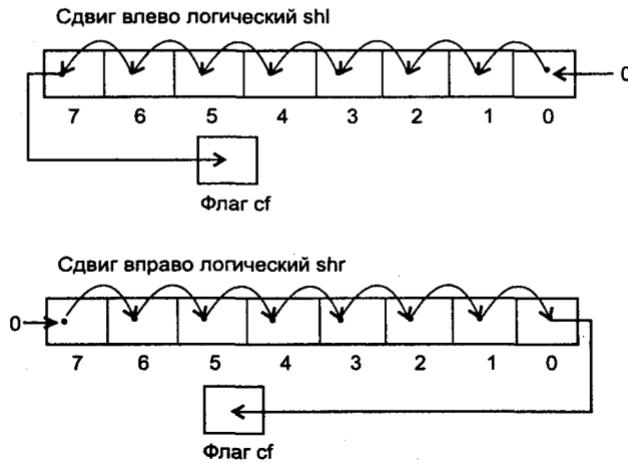


Рис. 9.2. Схема работы команд линейного логического сдвига

Ниже показан фрагмент программы, который выполняет преобразование двух неупакованных BCD-чисел в слове памяти `bcd_dig` в упакованное BCD-число в регистре `a1`.

```
...
bcd_dig dw 0905h ;описание неупакованного BCD-числа 95
...
mov ax,bcd_dig ;пересылка
shl ah,4 ;сдвиг влево
add a1,ah ;сложение для получения результата: a1=95h
```

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом работают со знаковым разрядом операнда:

`sal` *операнд, счетчик_сдвигов* (Shift Arithmetic Left) — арифметический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением *счетчик_сдвигов*. Справа (в позицию младшего бита) вписываются нули. Команда `sal` не сохраняет знака, но устанавливает флаг `sf` в случае смены знака очередным выдвинутым битом. В остальном команда `sal` полностью аналогична команде `shl`;

`sar` *операнд, счетчик_сдвигов* (Shift Arithmetic Right) — арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением *счетчик_сдвигов*. Слева в операнд вписываются нули. Команда `sar` сохраняет знак, восстанавливая его после сдвига каждого очередного бита.

На рис. 9.3 показан принцип работы команд линейного арифметического сдвига.

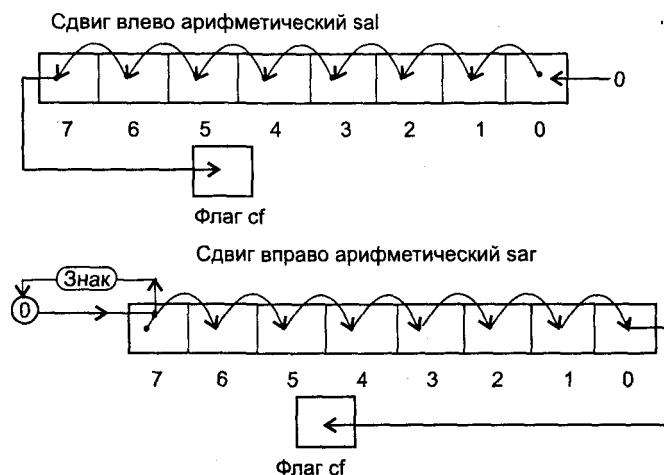


Рис. 9.3. Схема работы команд линейного арифметического сдвига

Команды арифметического сдвига позволяют выполнить умножение и деление операнда на степени двойки. Посмотрите на двоичное представление чисел 75 и 150:

Второе число является сдвинутым влево на один разряд первым числом. Если у вас еще есть сомнения, проделайте несколько умножений на 2, 4, 8 и т. д.

Аналогичная ситуация — с операцией деления. Сдвигая вправо операнд, мы, фактически, осуществляем операцию деления на степени двойки 2, 4, 8 и т. д.

Преимущество этих команд, по сравнению с командами умножения и деления, — в скорости их исполнения микропроцессором, что может пригодиться при оптимизации программы.

Циклический сдвиг

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

- команды простого циклического сдвига (рис. 9.4);
- команды циклического сдвига через флаг переноса cf (рис. 9.5).

К командам простого циклического сдвига относятся:

rol *операнд, счетчик_сдвигов* (Rotate Left) — циклический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые влево биты записываются в тот же operand справа;

ror *операнд, счетчик_сдвигов* — (Rotate Right) циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые вправо биты записываются в тот же operand слева.

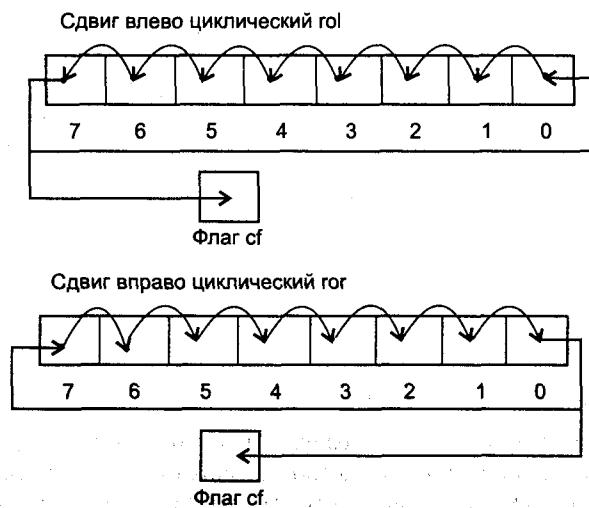


Рис. 9.4. Схема работы команд простого циклического сдвига

Как видно из рис. 9.4, команды простого циклического сдвига в процессе своей работы осуществляют одно полезное действие, а именно: циклически сдвигаемый бит не только вдвигается в operand с другого конца, но и одновременно его значение становится значением флага cf. К примеру, для того чтобы обменять содержимое двух половинок регистра eax, достаточно выполнить следующую последовательность команд:

```
mov    eax, ffff0000h  
mov    cl, 16  
rol    eax, cl
```

Команды циклического сдвига через флаг переноса cf отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в operand с другого его конца, а записывается сначала во флаг переноса cf. Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита в другой конец операнда (см. рис. 9.5). К командам циклического сдвига через флаг переноса cf относятся следующие:

rcl *операнд, счетчик_сдвигов* (Rotate through Carry Left) – циклический сдвиг влево через перенос. Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые биты поочередно становятся значением флага переноса cf;

rcr *операнд, счетчик_сдвигов* (Rotate through Carry Right) – циклический сдвиг вправо через перенос. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые биты поочередно становятся значением флага переноса cf.

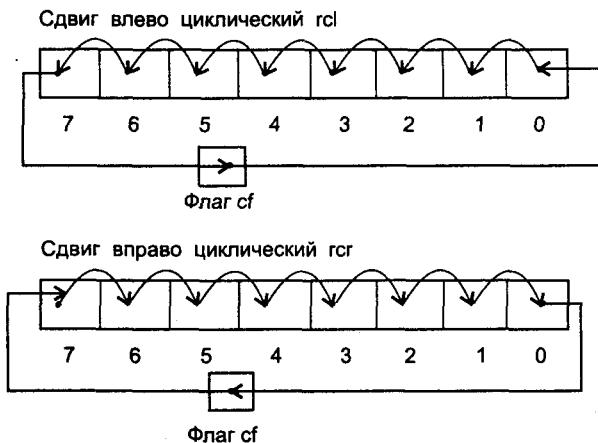


Рис. 9.5. Команды циклического сдвига через флаг переноса cf

Из рис. 9.5 видно, что при сдвиге через флаг переноса появляется промежуточный элемент, с помощью которого, в частности, можно производить подмену циклически сдвигаемых битов, в частности, рассогласование битовых последовательностей. Под рассогласованием битовой последовательности здесь и далее

подразумевается действие, которое позволяет некоторым образом локализовать и извлечь нужные участки этой последовательности и записать их в другое место. Например, рассмотрим, как переписать в регистр `bx` старшую половину регистра `eax` с одновременным ее обнулением в регистре `eax`:

```
    mov  cx,16      ;кол-во сдвигов для eax
m1:
    clc            ;сброс флага cf в 0
    rcl  eax,1      ;сдвиг крайнего левого бита из eax в cf
    rcl  bx          ;перемещение бита из cf справа в bx
    loop m1         ;цикл 16 раз
    rol  eax,16     ;восстановить правую часть eax
```

Команды простого циклического сдвига можно использовать для операций другого рода. К примеру, подсчитаем количество единичных битов в регистре `eax`:

```
xor  dx,dx      ;очистка dx для подсчета единичных бит
    mov  cx,32      ;число циклов подсчета
cyc1:
    jnc  not_one    ;метка цикла
    ror  eax,1      ;циклический сдвиг вправо на 1 бит
    inc  dx          ;переход, если очередной бит в cf
    not  not_one     ;не равен единице
    inc  dx          ;увеличение счетчика цикла
not_one:
    loop cyc1       ;переход на метку cyc1, если
                    ;значение в cx не равно 0
```

Этот фрагмент не требует особых пояснений, единственное, что нужно помнить, — особенности работы команды цикла `loop`. В полном объеме она будет рассмотрена на следующем уроке. Команда `loop` сравнивает значение регистра `cx` с нулем и, если оно не равно нулю, выполняет уменьшение `cx` на единицу и передачу управления на метку в программе, указанную в этой команде в качестве операнда.

Дополнительные команды сдвига

Система команд последних моделей микропроцессоров Intel, начиная с 80386, содержит дополнительные команды сдвига, расширяющие возможности, рассмотренные нами ранее. Это — *команды сдвигов двойной точности*:

`shld операнд_1, операнд_2, счетчик_сдвигов` — сдвиг влево двойной точности.

Команда `shld` производит замену путем сдвига битов операнда `операнд_1` влево, заполняя его биты справа значениями битов, вытесняемых из `операнд_2` согласно схеме на рис. 9.6. Количество сдвигаемых бит определяется значением `счетчик_сдвигов`, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственным операндом или содержаться в регистре `cl`. Значение `операнд_2` не изменяется;

`shrd операнд_1, операнд_2, счетчик_сдвигов` — сдвиг вправо двойной точности. Команда производит замену путем сдвига битов операнда `операнд_1` вправо,

заполняя его биты слева значениями битов, вытесняемых из *операнд_2* согласно схеме на рис. 9.7. Количество сдвигаемых бит определяется значением *счетчик_сдвигов*, которое может лежать в диапазоне 0..31. Это значение может задаваться непосредственным операндом или содержаться в регистре *c1*. Значение *операнд_2* не изменяется.



Рис. 9.6. Схема работы команды shld



Рис. 9.7. Схема работы команды shrд

Как мы отметили, команды *shld* и *shrд* осуществляют сдвиги до 32 разрядов, но за счет особенностей задания операндов и алгоритма работы эти команды можно использовать для работы с полями длиной до 64 бит. Например, рассмотрим, как можно осуществить сдвиг влево на 16 битов поля из 64 битов.

```
...
.data
pole_1 dd 0b21187f5h
pole_h dd 45ff6711h
.code
;...
.386
    mov c1,16 ;загрузка счетчика сдвига в c1
    mov eax,pole_h
    shr pole_1,eax,c1
    shr pole_h,c1 ;pole_1=87f50000h;
pole_h=6711b211h
```

Рассмотрим еще некоторые наиболее типичные примеры применения этих команд. Отметим следующий момент. Рассмотренные ниже действия, конечно, можно выполнить и множеством других способов, но эти являются самыми быстрыми. Если ваши программы должны работать максимально быстро, то есть смысл потратить время на разбор этих примеров.

Примеры работы с битовыми строками

Рассогласование битовых строк

Наглядный пример рассогласования последовательностей бит — преобразование неупакованного BCD-числа в упакованное BCD-число. Один из вариантов такого преобразования был рассмотрен нами выше при обсуждении команды линейного сдвига `shl`. Попробуем выполнить подобное преобразование с использованием команд сдвига двойной точности (листинг 9.2). В общем случае длина числа может быть произвольной, но при этом нужно учитывать ограничения, которые налагаются используемыми ресурсами микропроцессора. Ограничения связаны в основном с тем, что центральное место в преобразовании занимает регистр `eax`, поэтому, если преобразуемое число имеет размер более четырех байт, то его придется делить на части. Но это уже чисто алгоритмическая задача, поэтому в нашем случае предполагается, что неупакованное BCD-число имеет длину 4 байта.

Листинг 9.2. Преобразование BCD-числа (вариант 2)

```
:prg_9_2.asm
masm
model    small
stack     256
.data
len=4          ;длина неупакованного BCD-числа
unpck_BCD  label dword
dig_BCD   db    2,4,3,6 ;неупакованное BCD-число 6342
pck_BCD   dd    0      ;pck_BCD=00006342
.code
main:           ;точка входа в программу
    mov    ax,@data
    mov    ds,ax
    xor    ax,ax
    mov    cx,len
.386            ;это обязательно
    mov    eax,unpck_BCD
m1:
    shl    eax,4      ;убираем нулевую тетраду
    shld   pck_BCD,eax,4 ;тетраду с цифрой заносим в поле pck_BCD
    shl    eax,4      ;убираем тетраду с цифрой из eax
    loop   m1         ;цикл
    exit:           ;pck_BCD=00006342
    mov    ax,4c00h
    int    21h
end   main
```

Команды сдвига двойной точности `shld` и `shrd` позволяют осуществлять с максимально возможной скоростью вставку битовой строки из регистра в произвольное место другой (большей) строки бит в памяти и извлечение битовой подстроки из некоторой строки бит в памяти в регистр. В результате этих операций смежные с подстрокой биты по ее обеим сторонам остаются неизменными.

Вставка битовых строк

Рассмотрим пример вставки битовой строки длиной 16 бит, находящейся в регистре eax, в строку памяти str, начиная с ее 8 бита (листинг 9.3). Вставляемая битовая строка выровнена к левому краю регистра eax.

Листинг 9.3. Вставка битовой строки

```
<1>      ;prg_9_3.asm
<2>      masm
<3>      model small
<4>      stack 256
<5>      .data
<6>      bit_str    dd 11010111h ;строка для вставки
<7>      p_str dd 0ffff0000h ;вставляемая подстрока 0ffffh
<8>      .code
<9>      main: ;точка входа в программу
<10>         mov ax,@data
<11>         mov ds,ax
<12>         xor ax,ax
<13>         .386           ;это обязательно
<14>         mov eax,p_str
<15> ;правый край места вставки циклически переместить к краю
<16> ;строки bit_str (сохранение правого контекста):
<17>         ror bit_str,8
<18>         shr bit_str,16 ;сдвинуть строку вправо на длину подстроки (16 бит)
<19>         shld bit_str,eax,16 ;сдвинуть 16 бит
<20>         rol bit_str,8 ;восстановить младшие 8 бит
<21>         ...
<22>         exit:          ;bit_str=11fffff11
<23>         mov ax,4c00h
<24>         int 21h
<25>     end main
```

Листинг 9.3 удобно исследовать в отладчике. При его рассмотрении важно понять закономерность между непосредственными значениями, которые используются в командах строк 17–20, и теми значениями, которые присутствуют в постановке задачи. Общая методика такой вставки заключается в следующем:

- подогнать к правому краю строки младший бит места вставки в этой строке. Делать это нужно командой циклического сдвига, чтобы сохранить правую часть исходной строки. Величина сдвига определяется очень просто — это номер начальной позиции места вставки (см. листинг 9.3, строка 17);
- сдвинуть исходную строку вправо на количество битов, равное длине вставляемой подстроки (строка 18). Эти биты нам больше не нужны, поэтому для сдвига используется команда простого сдвига shr;
- командой shld вставить вставляемую подстроку в исходную подстроку. Перед этим, естественно, левый край вставляемой подстроки находится у левого края регистра eax (строка 19);
- восстановить командой циклического сдвига правую часть исходной строки (строка 20).

Наибольшей эффективности при использовании этой программы можно достичь, если оформить используемую в ней последовательность команд в виде макрокоманды. Понятие макрокоманды будет рассматриваться нами на уроке 13, но сейчас важно отметить, что в данном случае она позволит нам не задумываться о настройке строк 17–20 на конкретную вставку. При изучении материала урока 13 вы можете поэкспериментировать с данной программой, разработав на ее основе макрокоманду.

Извлечение битовых строк

Рассмотрим пример извлечения 16 битов из строки в памяти `bit_str`, начиная с бита 8, в регистр `eax` (листинг 9.4). Результат следует выровнять по правому краю регистра `eax`; строка `bit_str` не изменяется. Этот пример можно рассматривать как обратный тому, который мы только что привели в листинге 9.3. Методика извлечения битовой подстроки, если вы разобрались с программой вставки битовой строки, не должна вызвать у вас трудностей.

Листинг 9.4. Извлечение битовой строки

```
:prg_9_4.asm
masm
model    small
stack     256
.data
bit_str  dd      11fffff11h    ;строка для извлечения
.code
main:   ;точка входа в программу
        mov    ax,@data
        mov    ds,ax
        xor    ax,ax
.386          ;это обязательно
;левый край места извлечения циклически переместить к левому краю
;строки bit_str (сохранение левого контекста)
        rol    bit_str,8
        mov    ebx,bit_str ;подготовленную строку в ebx
        shld   eax,ebx,16 ;вставить извлекаемые 16 бит
;в регистр eax
        ror    bit_str,8 ;восстановить старшие 8 бит
...
exit:           ;eax=0000ffff
        mov    ax,4c00h
        int    21h
end main
```

Пересылка битов

По сути, эта программа будет являться комбинацией двух предыдущих. Поэтому попробуйте самостоятельно разработать программу пересылки блока битов из одной битовой строки в другую, взяв за основу только что рассмотренные примеры (см. листинги 9.3 и 9.4).

Подведем некоторые итоги:

- Минимально адресуемая единица данных в микропроцессоре – байт. Логические команды позволяют манипулировать отдельными битами. Это единственные команды в системе команд микропроцессора, которые позволяют работать на битовом уровне. Этим, в частности, объясняется их важность.
- Возможность работы на битовом уровне позволяет в отдельных случаях существенно сэкономить память, особенно при моделировании различных массивов, содержащих одноразрядные флаги или переключатели.
- Команды сдвига позволяют выполнять быстрое умножение и деление операндов на степени двойки, а также эффективное преобразование данных.
- Применение команд циклического сдвига и сдвига двойной точности позволяет реализовать максимально быстрые операции по рассогласованию, перемещению, вставке и извлечению битовых подстрок.

10

УРОК

Команды передачи управления

-
- Программирование нелинейных алгоритмов
 - Классификация команд передачи управления
 - Команды безусловной передачи управления
 - Понятие процедуры в языке ассемблера
 - Команды условной передачи управления
 - Средства организации циклов в языке ассемблера
-

На предыдущем уроке мы познакомились с некоторыми командами, из которых формируются линейные участки программы. Каждая из них в общем случае выполняет некоторые действия по преобразованию или пересылке данных, после чего микропроцессор передает управление следующей команде. Но очень мало программ работают таким последовательным образом. Обычно в программе есть точки, в которых нужно принять решение о том, какая команда будет выполнять следующей. Это решение может быть:

- **безусловным** — в данной точке необходимо передать управление не той команде, которая идет следующей, а другой, которая находится на некотором удалении от текущей команды;
- **условным** — решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных.

Как вы помните, программа представляет собой последовательность команд и данных, занимающих определенное пространство оперативной памяти. Это пространство памяти может быть либо непрерывным, либо состоять из нескольких фрагментов. На уроке 5 нами были рассмотрены средства организации фрагментации кода программы и ее данных на сегменты. То, какая команда программы должна выполняться следующей, микропроцессор узнает по содержимому пары регистров **cs:(e)ip**¹, в которой:

- **cs** — сегментный регистр кода, в котором находится физический (базовый) адрес текущего сегмента кода;
- **eip/ip** — регистр указателя команды, в котором находится значение, представляющее собой смещение в памяти следующей команды, подлежащей выполнению, относительно начала текущего сегмента кода. Напомню, почему мы записываем регистры **eip/ip** через косую черту. Какой конкретно регистр будет использоваться, зависит от установленных режимов адресации **use16** или **use32**. Если указано **use16**, то используется **ip**, если **use32**, то используется **eip**.

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

¹ При обсуждении архитектуры микропроцессора мы говорили, что команды извлекаются из памяти заранее в так называемый конвейер, поэтому адрес подлежащей выборке команды из памяти и содержимое пары **cs:(ip)** не одно и то же. Эта пара регистров содержит адрес команды в программе, которая будет выполняться следующей, а не той команды, которая будет выбираться на конвейер.

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

1. *Команды безусловной передачи управления:*

- команда безусловного перехода;
- вызов процедуры и возврата из процедуры;
- вызов программных прерываний и возврат из программных прерываний.

2. *Команды условной передачи управления:*

- команды перехода по результату команды сравнения;
- команды перехода по состоянию определенного флага;
- команды перехода по содержимому регистра esх/сх.

3. *Команды управления циклом:*

- команда организации цикла со счетчиком esх/сх;
- команда организации цикла со счетчиком esх/сх с возможностью досрочного выхода из цикла по дополнительному условию.

Возникает вопрос о том, каким образом обозначается то место, куда необходимо передать управление. В языке ассемблера это делается с помощью меток. *Метка* — это символическое имя, обозначающее определенную ячейку памяти, предназначенное для использования в качестве операнда в командах передачи управления.

Подобно переменной, транслятор ассемблера присваивает любой метке три атрибута:

- имя сегмента кода*, где эта метка описана;
- смещение* — расстояние в байтах от начала сегмента кода, в котором описана метка;
- тип метки, или атрибут расстояния*.

Последний атрибут может принимать два значения:

- near* — переход на эту метку возможен только в пределах сегмента кода, где эта метка описана. Физически это означает, что для перехода на метку достаточно изменить только содержимое регистра eip/ip;
- far* — переход на эту метку возможен только в результате межсегментной передачи управления, для осуществления которой требуется изменение содержимого как регистра eip/ip, так и регистра cs.

Метку можно определить двумя способами:

- оператором : (двоеточие);
- директивой label.

Синтаксис первого способа показан на рис. 10.1.

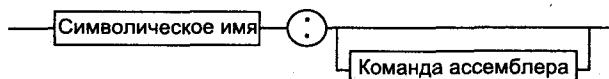


Рис. 10.1. Синтаксис описания метки оператором :

С помощью этого способа можно определить метку только ближнего типа — `near`. Символическое имя в программе может быть определено только один раз. Определенную таким образом метку можно использовать в качестве операнда в командах условного перехода `jcc` и безусловного перехода `jmp, call`. Эти команды, естественно, должны быть в сегменте кода, где определена метка. Команда ассемблера может находиться как на одной строке с меткой, так и на следующей.

Второй способ определения меток в программе использует директиву `label` (рис. 10.2).

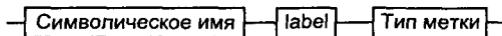


Рис. 10.2. Синтаксис директивы `label`

На рис. 10.2 тип метки принимает значения `near` или `far`. Обычно директиву `label` используют для определения идентификатора заданного типа. Например, следующие описания меток ближнего типа эквивалентны:

```
m1:  
    mov     ax,pole_1  
и  
m1  label near  
    mov     ax,pole_1
```

Понятно, что метка может быть только одного типа — либо `near`, либо `far`. Если возникает необходимость использовать для одной и той же команды метку и дальнего, и ближнего типов, то в этом случае необходимо определить две метки, причем метку дальнего типа нужно описать, используя директиву `label`, как показано в следующем фрагменте:

```
;...  
public   m_far ;сделать метку m_far видимой  
;для внешних программ  
...  
m_far    label far   ;определение метки дальнего типа m_far  
m_near:  label near ;определение метки ближнего типа n_far  
    mov     ax,pole_1  
...
```

Определив для команды `mov ax,pole_1` две метки, можно организовывать переход на эту команду как из данного сегмента команд, так и из других сегментов команд, в том числе принадлежащих другим модулям. Для того чтобы сделать видимым извне имя метки `m_far`, применяется директива `public`. К более подробному описанию этой директивы мы еще вернемся.

Другой часто встречающийся случай применения директивы `label` — это организация доступа к одной области памяти, как к области, содержащей данные разных типов, например:

```
;...  
mas_b    label byte  
mas_w    dw      15 dup (?)  
...  
;в этом фрагменте оба идентификатора относятся к одной области  
;памяти и дают возможность работать с ней, используя разные
```

;имена, либо как с байтовым массивом, либо как с массивом слов

```
...    mov    mas_b+10,al ;запись из al в массив  
          ;байтов (в 11-й байт)  
...  
    mov    mas_w,ax ;запись из ax в первое  
          ;слово области mas_w  
...
```

Введем еще одно очень важное понятие ассемблера, имеющее прямое отношение к меткам, — *счетчик адреса команд*. Мы уже упоминали о нём на первых уроках и говорили, что транслятор ассемблера обрабатывает исходную программу, написанную пользователем, последовательно — команду за командой. При этом он ведет счетчик адреса команд, который для первой исполняемой команды равен нулю, а далее, по ходу обработки очередной команды транслятором, он увеличивается на длину этой команды. По сути, счетчик адреса команд — это смещение конкретной команды относительно начала сегмента кода. Таким образом, каждая команда во время трансляции имеет *адрес*, равный значению счетчика адреса команд. Обратитесь к уроку 4 и еще раз посмотрите на приведенный в нем листинг 4.1. Первая колонка в листинге — номер строки листинга. Вторая колонка (или третья, если присутствует колонка с уровнем вложенности) — смещение команды относительно начала сегмента кода или, как мы сейчас определили, счетчик адреса. Значение, на которое он увеличивается по мере обработки ассемблером очередной строки исходной программы, равно значению длины машинной команды в этой строке. Исходя из этого, ясно, почему счетчик адреса растет только после тех строк исходной программы, которые генерируют некоторое машинное представление (в том числе после директив резервирования и инициализации данных в сегменте данных).

Транслятор ассемблера обеспечивает нам две возможности работы с этим счетчиком:

- использование меток, атрибутику смещения которых транслятор присваивает значение счетчика адреса той команды, перед которой они появились;
- применение специального символа \$ для обозначения счетчика адреса команд. Этот символ позволяет в любом месте программы использовать численное значение счетчика адреса. Классический пример:

```
...  
.data  
;вычисление длины строки в сегменте данных  
Str_Mes db "Работаешь на ПК - изучи ассемблер"  
Len_Msg=$-Str_Mes  
...
```

После ассемблирования значение Len_Msg будет равно длине строки, так как значение символа \$ в месте его появления отличается от Str_Mes ровно на длину строки.

Кроме возможности получения значения счетчика адреса, компилятор TASM позволяет при необходимости установить счетчик адреса в нужное абсолютное значение. Это делается с помощью директивы ORG.

ORG выражение — задает значение счетчика адреса. *Выражение* должно быть таким, чтобы ассемблер мог преобразовать его к абсолютному числу при первом проходе трансляции.

К примеру, эту директиву всегда используют при создании исполняемого файла с типом .com. В контексте нашего обсуждения поясним, в чем здесь суть. Мы обсуждали сегментацию и разделение программы на сегменты. Программа в СОМ-формате состоит из одного сегмента величиной не более 64 Кбайт. Сегментные регистры cs и ds содержат одно и то же значение физического адреса, а ss указывает на конец этого единственного сегмента. Программа-загрузчик операционной системы, считывая с диска исполняемые файлы типов .exe и .com, производит определенные действия. В частности настраивает перемещаемые адреса программы на их конкретные физические значения. Кроме того, к началу каждой исполняемой программы в памяти добавляется специальная область величиной 256 байт (100h) — *префикс программного сегмента* (PSP). Он предназначен для хранения различной информации о загруженном исполняемом модуле. Для программ формата .com блок PSP находится в начале сегмента размером в 64 Кбайт. В исходной программе, для которой планируется формат исполняемого файла .com, мы должны предусмотреть место для блока PSP, что и делается директивой org 100h.

Чтобы закончить разговор о файлах этого типа, разберемся с тем, как получить исполняемый модуль формата .com. Трансляция программы выполняется как обычно. Далее возможны два варианта действий:

○ использование утилиты Tlink с опцией /t:

`tlink /t имя_объектного_файла`

Этот вариант подходит только в том случае, если вы правильно оформили исходный текст программы для формата .com. Кроме использования директивы org 100h, это предполагает:

- отсутствие разделения сегментов данных и стека, то есть данные описаны в сегменте кода и для их обхода необходимо использовать команду безусловного перехода jmp;
- использование директивы assume для указания транслятору на необходимость связать содержимое регистров ds и ss с сегментом кода;

```
codeseg segment para "code"
assume cs: codeseg,ds:codeseg,ss:codeseg
org 100h
jmp m1
;здесь описываем данные
m1:
;далее идут команды программы
...
```

○ использование специальной утилиты exe2bin. Эта утилита позволяет преобразовать уже полученный ранее исполняемый модуль в формат .exe в формат .com:

`exe2bin имя_файла_exe имя_файла_com.com`

Этот вариант не требует специального оформления исходного текста программы. Единственным требованием является то, чтобы исходный текст был достаточно мал по объему.

Безусловные переходы

Предыдущее обсуждение выявило некоторые детали механизма перехода. Команды перехода модифицируют регистр указателя команды `eip/ip` и, возможно, сегментный регистр кода `cs`. Что именно должно подвергнуться модификации, зависит:

- от типа операнда в команде безусловного перехода (ближний или дальний);
- от указания перед адресом перехода (в команде перехода) модификатора; при этом сам адрес перехода может находиться либо непосредственно в команде (прямой переход), либо в регистре или ячейке памяти (косвенный переход). Модификатор может принимать следующие значения:

`near ptr` — прямой переход на метку внутри текущего сегмента кода. Модифицируется только регистр `eip/ip` (в зависимости от заданного типа сегмента кода `use16` или `use32`) на основе указанных в команде адреса (метки) или выражения, использующего символ извлечения значения счетчика адреса команд — `$`.

`far ptr` — прямой переход на метку в другом сегменте кода. Адрес перехода задается в виде непосредственного операнда или адреса (метки) и состоит из 16-битного селектора и 16/32-битного смещения, которые загружаются, соответственно, в регистры `cs` и `ip/eip`.

`word ptr` — косвенный переход на метку внутри текущего сегмента кода. Модифицируется (значением смещения из памяти, по указанному в команде адресу или из регистра) только `eip/ip`. Размер смещения 16 или 32 бита.

`dword ptr` — косвенный переход на метку в другом сегменте кода. Модифицируются (значением из памяти — и только из памяти, из регистра нельзя) оба регистра, `cs` и `eip/ip`. Первое слово/двойное слово этого адреса представляет смещение и загружается в `ip/eip`; второе/третье слово загружается в `cs`.

Команда безусловного перехода `jmp`

Синтаксис команды безусловного перехода:

`jmp [модификатор] адрес_перехода` — безусловный переход без сохранения информации о точке возврата. `Адрес_перехода` представляет собой адрес в виде метки либо адрес области памяти, в которой находится указатель перехода.

Всего в системе команд микропроцессора есть несколько кодов машинных команд безусловного перехода `jmp`. Их различия определяются дальностью перехода и способом задания целевого адреса. Дальность перехода определяется

местоположением операнда *адрес перехода*. Этот адрес может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется *внутрисегментным*, или *близким*, во втором — *межсегментным*, или *дальним*.

Внутрисегментный переход предполагает, что изменяется только содержимое регистра *eip/ip*. Можно выделить три варианта внутрисегментного использования команды *jmp*:

- прямой короткий;
- прямой;
- косвенный.

Прямой короткий внутрисегментный переход применяется, когда расстояние от команды *jmp* до *адреса перехода* не более чем -128 или +127 байтов. В этом случае транслятор ассемблера формирует машинную команду безусловного перехода длиной всего 2 байта (размер обычной команды внутрисегментного безусловного перехода составляет 3 байта). Первый байт в этой команде — код операции, значение которого говорит о том, что микропроцессор должен особым образом трактовать второй байт команды. Значение второго байта вычисляется транслятором как разность между значением смещения команды, следующей за *jmp*, и значением адреса перехода. При осуществлении прямого короткого перехода нужно иметь в виду следующий тонкий момент, связанный с местоположением адреса перехода и самой команды *jmp*. Если адрес перехода расположен до команды *jmp*, то ассемблер формирует короткую команду безусловного перехода без дополнительных указаний. В случае расположения адреса перехода после команды *jmp* транслятор не может сам определить, что переход короткий, так как у него еще нет информации об адресе перехода. Для оказания помощи компилятору в формировании команды короткого безусловного перехода в дополнение к вышерассмотренным используют модификатор *short ptr*:

```
...
    jmp    short ptr m1
... ;не более 35-40 команд (127 байтов)
```

m1:

или

```
...
m1:
... ;не более 35-40 команд (-128 байтов)
    jmp    m1
...
```

Прямой внутрисегментный переход отличается от прямого короткого внутрисегментного перехода тем, что длина машинной команды *jmp* в этом случае составляет 3 байта. Увеличение длины связано с тем, что поле адреса перехода в машинной команде *jmp* расширяется до двух байт, а это, в свою очередь, позволяет производить переходы в пределах 64 Кбайт относительно следующей за *jmp* команды.

```
m1:
... ;расстояние более 128 байт и менее 64 Кбайт
```

```
jmp m1
```

Косвенный внутрисегментный переход подразумевает «косвенность» задания адреса перехода. Это означает, что в команде указывается не сам адрес перехода, а место, где он «лежит». Приведем несколько примеров, в которых двухбайтовый адрес перехода выбирается либо из регистра, либо из области памяти.

```
lea    bx,m1
jmp    bx      ;адрес перехода в регистре bx
...
m1:
...
.data
addr_m1 dw    m1
...
.code
;...
jmp    addr_m1          ;адрес перехода в ячейке памяти addr_m1
;...
m1:
```

Приведем еще несколько вариантов косвенного внутрисегментного перехода.

```
<1>    ...
<2>    .data
<3>    addr dw    m1
<4>          dw    m2
<5>    ...
<6>    .code
<7>    ...
<8>    cycl:
<9>        mov    si,0
<10>       jmp   addr[si]    ;адрес перехода в слове памяти addr+(si)
<11>       ...
<12>       mov    si,2
<13>       jmp   cycl
<14>       m1:
<15>       ...
<16>       m2:
<17>
<18>       ...
```

В этом примере одна команда `jmp` (строка 10) может производить переходы на разные метки. Выбор конкретной метки перехода определяется содержимым `si`. Операнд команды `jmp` определяет адрес перехода косвенно, после вычисления выражения `addr+(si)`.

```
<1>    ...
<2>    .data
<3>    addr dw    m1
<4>    ...
<5>    .code
<6>    ...
<7>    |     lea    si,addr
<8>    |     jmp   near ptr[si] ;адрес перехода в ячейке памяти addr
```

<9> ...
<10> m1:

В данном случае указание модификатора `near ptr` обязательно, так как, в отличие от предыдущего способа, адрес ячейки памяти `addr` с адресом перехода транслятору передается неявно (строки 3, 7 и 8) и, не имея информации о метке, он не может определить, какой именно вид перехода осуществляется — внутрисегментный или межсегментный. *Межсегментный переход* предполагает другой формат машинной команды `jmp`. При осуществлении межсегментного перехода кроме регистра `eip/ip` модифицируется также и регистр `cs`. Аналогично внутрисегментному переходу, межсегментный переход поддерживают два варианта команд безусловного перехода: прямой и косвенный.

Команда *прямого межсегментного перехода* имеет длину пять байтов, из которых два байта составляют значение смещения и два байта — значение сегментной составляющей адреса.

```
seg_1    segment
...
    jmp    far ptr m2 ;здесь far обязательно
...
m1 label far
...
seg_1    ends
seg_2    segment
...
m2 label far
    jmp    m1 ;здесь far необязательно
```

Рассматривая этот пример, обратите внимание на использование модификаторов `far ptr` в команде `jmp`. Обязательность их задания определяется все той же логикой работы однопроходного транслятора. Если описание метки встречается в исходном тексте программы раньше, чем соответствующая ей команда перехода (метка `m1`), то задание модификатора необязательно, так как транслятор все знает о данной метке и сформирует нужную пятибайтовую форму команды безусловного перехода. В случае когда команда перехода встречается до описания соответствующей метки, транслятор не имеет еще никакой информации о метке и модификатор `far ptr` в команде `jmp` опускать нельзя, так как транслятор не знает, какую форму команды формировать — трехбайтную или пятибайтную. Без специального указания модификатора транслятор будет формировать трехбайтную команду внутрисегментного перехода.

Команда *косвенного межсегментного перехода* в качестве операнда имеет адрес области памяти, в которой содержатся смещение и сегментная часть целевого адреса перехода.

```
data    segment
addr_m1 dd    m1 ;в поле addr_m1 значения смещения
           ;и адреса сегмента метки m1
data    ends
code_1   segment
...
    jmp    m1
...
```

```
code_1    ends
code_2    segment
...
m1 label far
    mov    ax,bx
...
code_2    ends
```

Как вариант косвенного межсегментного перехода необходимо отметить *косвенный регистровый межсегментный переход*. В этом виде перехода адрес перехода указывается косвенно; он содержится в регистре. Это очень удобно для программирования динамических переходов, в которых адрес перехода может изменяться на стадии выполнения программы.

```
data    segment
addr_m1 dd    m1      ; в поле addr_m1 значения смещения
          ; и адреса сегмента метки m1
data    ends
code_1   segment
...
    lea    bx,addr_m1
    jmp    dword ptr[bx]
...
code_1   ends
code_2   segment
...
m1 label far
    mov    ax,bx
...
code_2   ends
```

Таким образом, модификаторы `short ptr`, `near ptr` и `word ptr` применяются для организации внутрисегментных переходов, а `far ptr` и `dword ptr` — межсегментных.

Для полной ясности нужно еще раз подчеркнуть, что если тип сегмента `use32`, то в тех местах, где речь шла о регистре `ip`, можно использовать `eip` и, соответственно, размеры полей смещения увеличить до 32 битов.

Процедуры

До сих пор мы рассматривали примеры программ, предназначенные для однократного выполнения. Но приступив к программированию достаточно серьезной задачи, вы столкнетесь с тем, что у вас, наверняка, появятся повторяющиеся участки кода. Они могут быть небольшими, а могут занимать и достаточно много места. В последнем случае эти фрагменты будут существенно затруднять чтение текста программы, снижать ее наглядность, усложнять отладку и служить неисчерпаемым источником ошибок. В языке ассемблера есть несколько средств, решающих проблему дублирования участков программного кода. К ним относятся:

- механизм процедур;
- макроассемблер;
- механизм прерываний.

На данном уроке нами будут рассмотрены основы механизма процедур. Важность этого вопроса требует рассмотрения его в полном объеме на отдельном уроке (что мы и сделаем на уроке 14). Макроассемблер и прерывания также будут рассмотрены как отдельные вопросы (на уроках 13, 15 и 17).

Процедура, часто называемая также *подпрограммой*, — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры. Другими словами, процедуру можно определить как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: **PROC** и **ENDP**.

Синтаксис описания процедуры таков (рис. 10.3).

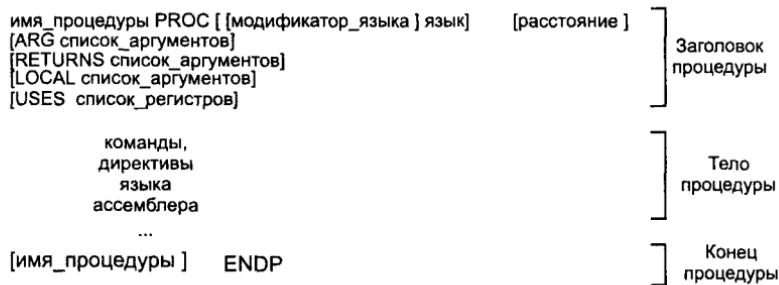


Рис. 10.3. Синтаксис описания процедуры в программе

Из рис. 10.3 видно, что в заголовке процедуры (директиве **PROC**) обязательным является только задание имени процедуры. Среди большого количества операндов директивы **PROC** следует особо выделить **[расстояние]**. Этот атрибут может принимать значения **near** или **far** и характеризует возможность обращения к процедуре из другого сегмента кода. По умолчанию атрибут **[расстояние]** принимает значение **near**.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока и, соответственно, будет осуществлять выполнение команд процедуры. Учитывая это обстоятельство, есть следующие варианты размещения процедуры в программе:

- в начале программы (до первой исполняемой команды);
- в конце (после команды, возвращающей управление операционной системе);
- промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы. В этом случае необходимо предусмотреть обход процедуры с помощью команды безусловного перехода **jmp**;
- в другом модуле.

Размещение процедуры в начале сегмента кода предполагает, что последовательность команд, ограниченная парой директив `PROC` и `ENDP`, будет размещена до метки, обозначающей первую команду, с которой начинается выполнение программы. Эта метка должна быть указана как параметр директивы `END`, обозначающей конец программы:

```
model    small
.stack   100h
.data
.code
my_proc proc near
...
ret
my_proc endp
start:
...
endstart
```

Обявление имени процедуры в программе равнозначно объявлению метки, поэтому директиву `PROC` в частном случае можно рассматривать как завуалированную форму определения метки в программе. Поэтому сама исполняемая программа также может быть оформлена в виде процедуры, что довольно часто и делается с целью пометить первую команду программы, с которой должно начаться выполнение. При этом не забывайте, что имя этой процедуры нужно обязательно указывать в заключительной директиве `END`. Такой синтаксис мы уже неоднократно использовали в своих программах. Так, последний рассмотренный фрагмент будет абсолютно эквивалентен следующему:

```
model    small
.stack   100h
.data
.code
my_proc proc near
...
ret
my_proc endp
start    proc
...
start    endp
endstart
```

В этом фрагменте после загрузки программы в память управление будет передано первой команде процедуры с именем `main`.

Размещение процедуры в конце программы предполагает, что последовательность команд, ограниченная директивами `PROC` и `ENDP`, будет размещена после команды, возвращающей управление операционной системе.

```
model    small
.stack   100h
.data
.code
start:
...
mov     ax,4c00h
```

```
int 21h ;возврат управления операционной системе
my_proc proc near
...
ret
my_proc endp
endstart
```

Промежуточный вариант расположения тела процедуры предполагает ее размещение внутри другой процедуры или основной программы. В этом случае необходимо предусмотреть обход тела процедуры, ограниченного директивами PROC и ENDP, с помощью команды безусловного перехода jmp:

```
model small
.stack 100h
.data
.code
start:
...
jmp m1
my_proc proc near
...
ret
my_proc endp
m1:
...
mov ax,4c00h
int 21h ;возврат управления операционной системе
endstart
```

Последний вариант расположения описаний процедур — в *отдельном сегменте кода* — предполагает, что часто используемые процедуры выносятся в отдельный файл. Файл с процедурами должен быть оформлен как обычный исходный файл и подвергнут трансляции для получения объектного кода. Впоследствии этот объектный файл утилитой TLINK можно объединить с файлом, где процедуры используются. Использование утилиты TLINK описано на уроке 4. Этот способ предполагает наличие в исходном тексте программы еще некоторых элементов, связанных с особенностями реализации концепции модульного программирования в языке ассемблера. Поэтому в полном объеме этот способ будет рассмотрен на уроке 14.

Как обратиться к процедуре? Так как имя процедуры обладает теми же атрибутами, что и обычная метка в команде перехода, то обратиться к процедуре, в принципе, можно с помощью любой команды перехода. Но есть одно важное свойство, которое можно использовать благодаря специальному *механизму вызова процедур*. Суть состоит в возможности сохранения информации о контексте программы в точке вызова процедуры. Под *контекстом* понимается информация о состоянии программы в точке вызова процедуры. В системе команд микропроцессора есть две команды, осуществляющие работу с контекстом. Это команды call и ret:

call [*модификатор*] *имя_процедуры* — вызов процедуры (подпрограммы). Команда call, подобно jmp, передает управление по адресу с символическим именем *имя_процедуры*, но при этом в стеке сохраняется адрес возврата. Адрес возврата — это адрес команды, следующей после команды call.

`ret [число]` — возвратить управление вызывающей программе. Команда `ret` считывает адрес возврата из стека и загружает его в регистры `cs` и `ip/eip`, тем самым возвращая управление на команду, следующую в программе за командой `call`. `[число]` — необязательный параметр, обозначающий количество элементов, удаляемых из стека при возврате из процедуры. Размер элемента определяется хорошо знакомыми нам параметрами директивы `segment` — `use16` или `use32` (или соответствующим параметром упрощенных директив сегментации). Если указано `use16`, то `[число]` — это значение в байтах; если `use32` — в словах.

Для команды `call`, как и для `jmp`, актуальна проблема организации ближних и дальних переходов. Это видно из формата команды, где присутствует `[модификатор]`. Как и в случае команды `jmp`, вызов процедуры командой `call` может быть:

- **внутрисегментным** — процедура находится в текущем сегменте кода (имеет тип `near`), и в качестве адреса возврата команда `call` сохраняет только содержимое регистра `ip/eip`, что вполне достаточно для осуществления возврата (рис. 10.4);

Процедура ближнего типа `my_proc`:

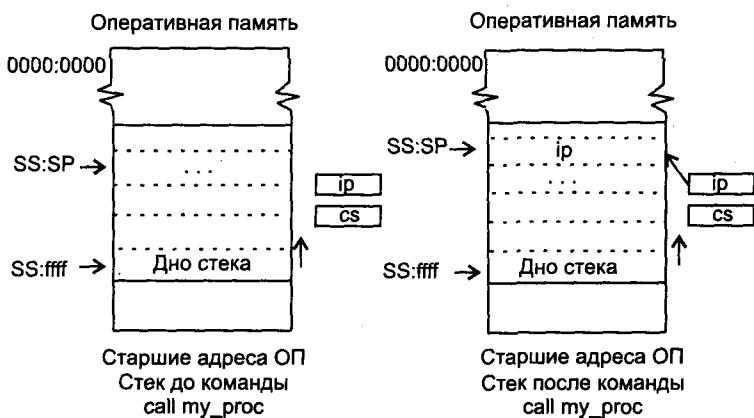


Рис. 10.4. Содержимое стека до и после команды вызова процедуры ближнего типа

- **межсегментным** — процедура находится в другом сегменте кода (имеет тип `far`), и для осуществления возврата команда `call` должна запомнить содержимое обоих регистров, `cs` и `ip/eip`. Очередность размещения их в стеке такова: сначала `cs`, затем `ip/eip` (рис. 10.5).

Важно отметить, что одна и та же процедура не может быть процедурой и ближнего, и дальнего типов. Таким образом, если процедура используется в текущем сегменте кода, но может вызываться и из другого сегмента программы, то она должна быть объявлена процедурой типа `far`. Подобно команде `jmp`, существуют четыре разновидности команды `call`. Какая именно команда будет сформирована, зависит от значения `[модификатор]` в команде вызова процедуры `call` и атрибута дальности в описании процедуры. Если процедура описана в начале сегмента данных с указанием дальности в ее заголовке, то при ее вызове `[модификатор]` можно не указывать: транслятор сам разберется, какую команду `call` ему нужно фор-

мировать. Если же процедура описана после ее вызова, например, в конце текущего сегмента или в другом сегменте, то при ее вызове нужно указать ассемблеру тип вызова, чтобы он мог за один проход правильно сформировать команду `call`. Значения [модификатор] такие же, как и у команды `jmp`, за исключением `short ptr`.

Процедура дальнего типа `my_proc`:

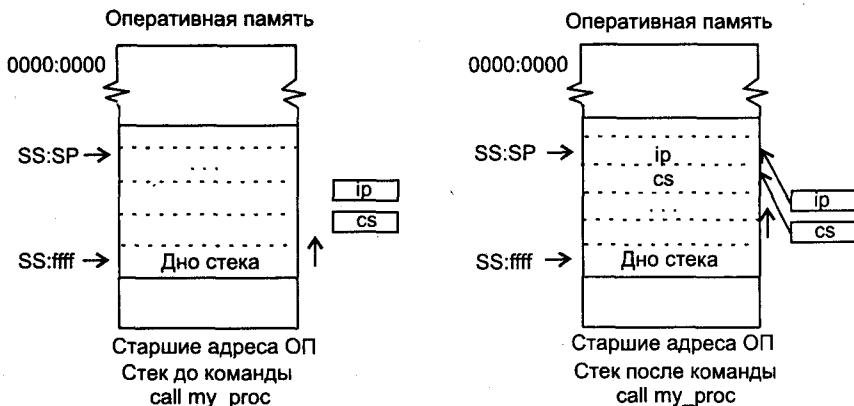


Рис. 10.5. Содержимое стека до и после команды вызова процедуры дальнего типа

Последний и, наверное, самый важный вопрос, возникающий при работе с процедурами, — как правильно передать параметры процедуре и вернуть результат? Этот вопрос тесно связан с концепцией модульного программирования и подробно будет рассматриваться на уроке 14. Примеры использования процедур вы можете посмотреть в листингах подпрограмм, предназначенных для вычисления четырех основных арифметических действий с двоичными и десятичными (BCD) числами и находящихся на дискеете в каталоге урока 8.

Условные переходы

До сих пор мы рассматривали команды перехода с «безусловным» принципом действия, но в системе команд микропроцессора есть большая группа команд, умеющих принимать решение о том, какая команда должна выполняться следующей. Решение принимается в зависимости от определенных условий. Условие определяется выбором конкретной команды перехода. Микропроцессор имеет 18 команд условного перехода, позволяющие проверить:

- отношение между operandами со знаком («больше-меньше»);
- отношение между operandами без знака («выше-ниже»)¹;
- состояниями арифметических флагов zf, sf, cf, of, pf (но не af).

¹ Термины «больше-меньше» и «выше-ниже» происходят от соответствующих английских терминов «greater-less» и «above-below». Первые буквы этих терминов входят в состав мемориических обозначений соответствующих команд условного перехода. Несмотря на кажущуюся синонимичность этих терминов, на самом деле они отражают тот факт, что соответствующие им команды условного перехода анализируют разные флаги. Смотрите также далее пояснения в тексте.

Команды условного перехода имеют одинаковый синтаксис:

jcc метка_перехода

Как видно, мнемокод всех команд начинается с «j» — от слова *jump* (прыжок), *сс* — определяет конкретное условие, анализируемое командой. Что касается операнда *метка_перехода*, то эта метка может находиться только в пределах текущего сегмента кода; межсегментной передачи управления в условных переходах не допускается. В связи с этим отпадает вопрос о модификаторе, который присутствовал в синтаксисе команд безусловного перехода. В ранних моделях микропроцессора (8086, 80186 и 80286) команды условного перехода могли осуществлять только короткие переходы — на расстояние от -128 до +127 байтов от команды, следующей за командой условного перехода. Начиная с модели микропроцессора 80386, это ограничение снято, но, как видите, только в пределах текущего сегмента кода.

Для того чтобы принять решение о том, куда будет передано управление командой условного перехода, предварительно должно быть сформировано условие, на основании которого и будет приниматься решение о передаче управления. Источниками такого условия могут быть:

- любая команда, изменяющая состояние арифметических флагов;
- команда сравнения *cmp*, сравнивающая значения двух operandов;
- состояние регистра *есх/cx*.

Обсудим эти варианты, чтобы разобраться с тем, как работают команды условного перехода.

Команда сравнения *cmp*

Команда сравнения *cmp* имеет интересный принцип работы. Он абсолютно такой же, как и у команды вычитания *sub* *операнд_1, операнд_2*. Команда *cmp* так же, как и команда *sub*, выполняет вычитание operandов и устанавливает флаги. Единственное, чего она не делает, — это запись результата вычитания на место первого операнда.

Синтаксис команды *cmp*:

cmp операнд_1, операнд_2 (*compare*) — сравнивает два операнда и по результатам сравнения устанавливает флаги.

Флаги, устанавливаемые командой *cmp*, можно анализировать специальными командами условного перехода. Прежде чем мы их рассмотрим, уделим немногого внимания мнемонике этих команд условного перехода (табл. 10.1). Понимание обозначений при формировании названия команд условного перехода (элемент в названии команды *jcc*, обозначенный нами *сс*) облегчит их запоминание и дальнейшее практическое использование.

Не удивляйтесь тому обстоятельству, что одинаковым значениям флагов соответствуют несколько разных мнемокодов команд условного перехода (они отделены друг от друга косой чертой в табл. 10.2). Разница в названии обусловлена желанием разработчиков микропроцессора облегчить использование команд условного перехода в сочетании с определенными группами команд. Поэтому разные названия отражают, скорее, различную функциональную направленность. Тем

не менее то, что эти команды реагируют на одни и те же флаги, делает их абсолютно эквивалентными и равноправными в программе. Поэтому в табл. 10.2 они сгруппированы не по названиям, а по значениям флагов (условиям), на которые они реагируют.

Таблица 10.1. Значение аббревиатур в названии команды jcc

| Мнемоническое обозначение | Английский | Русский | Тип operandов |
|---------------------------|------------|-------------------------|-----------------|
| E e | equal | Равно | Любые |
| N n | not | Не | Любые |
| G g | greater | Больше | Числа со знаком |
| L l | less | Меньше | Числа со знаком |
| A a | above | Выше, в смысле «больше» | Числа без знака |
| B b | below | Ниже, в смысле «меньше» | Числа без знака |

Таблица 10.2. Перечень команд условного перехода для команды cmpr операнд_1,операнд_2

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

В качестве примера применения команды cmp рассмотрим фрагмент программы, который обнуляет поле pole_m длиной n байт:

```
model small
.stack 100h
.data
n equ 50
pole_m db n dup (?)
.code
start:
...
xor bx,bx ;bx=0
```

```

m1: mov    mem[bx],0
      inc    bx
      cmp    bx,n
      jne    m1
exit:
      mov    ax,4c00h
      int    21h           ;возврат управления операционной системе
endstart

```

Так как команды условного перехода не изменяют флагов, то после одной команды `cmp` вполне могут следовать несколько команд условного перехода. Это может быть сделано для того, например, чтобы исследовать каждую из альтернативных ветвей: «больше», «меньше» или «равно»:

```

.data
masdb  dup (?)
.code
...
      cmp    mas[si],5      ;сравнить очередной элемент
      ; массива с 5
      je     eq1            ;переход, если элемент mas равен 5
      jl     low             ;переход, если элемент mas меньше 5
      jg     grt             ;переход, если элемент mas больше 5
eq1:
...
low:
...
grt:
...

```

Команды условного перехода и флаги

Мнемоническое обозначение некоторых команд условного перехода отражает название флага, с которым они работают, и имеет следующую структуру: первой буквой идет символ «`j`» (jump, переход), вторым — либо обозначение флага, либо символ отрицания «`n`», после которого стоит название флага. Такая структура команды отражает ее назначение. Если символа «`n`» нет, то проверяется состояние флага, и, если он равен 1, производится переход на метку перехода. Если символ «`n`» присутствует, то проверяется состояние флага на равенство 0, и в случае успеха производится переход на метку перехода. Мнемокоды команд, названия флагов и условия переходов приведены в табл. 10.3. Эти команды можно использовать после любых команд, изменяющих указанные флаги.

Если внимательно посмотреть на табл. 10.2 и 10.3, видно, что многие команды условного перехода в них являются эквивалентными, так как в основе их, и других лежит анализ одинаковых флагов.

В листинге 10.1 приведен пример программы, производящей замену в строке символов длиной n байт малых (строчных) букв английского алфавита на большие (прописные). Для осмысленного рассмотрения этого примера вспомним коды ASCII, соответствующие этим буквам. Строчные и прописные буквы в таблице ASCII упорядочены по алфавиту. Строчным буквам в этой таблице соответствует диапазон кодов 61h–7ah, а прописным — 41h–5ah. Для того чтобы понять

идею, лежащую в основе алгоритма преобразования, достаточно сравнить представления соответствующих прописных и строчных букв в двоичном виде:

a – 0110 0001 ... z – 0111 1010

A – 0100 0001 ... Z – 0101 1010

Как видно из приведенного двоичного представления, для выполнения преобразования между строчными и прописными буквами достаточно всего лишь инвертировать 5-й бит.

Таблица 10.3. Команды условного перехода и флаги

| Название флага | Номер бита в eflags	flags | Команда условного перехода | Значение флага для осуществления перехода |
|----------------------|------------------------------|-------------------------------|--|
| Флаг переноса cf | 1 | jc | cf = 1 |
| Флаг четности pf | 2 | jp | pf = 1 |
| Флаг нуля zf | 6 | jz | zf = 1 |
| Флаг знака sf | 7 | js | sf = 1 |
| Флаг переполнения of | 11 | jo | of = 1 |
| Флаг переноса cf | 1 | jnc | cf = 0 |
| Флаг четности pf | 2 | jnp | pf = 0 |
| Флаг нуля zf | 6 | jnz | zf = 0 |
| Флаг знака sf | 7 | jns | sf = 0 |
| Флаг переполнения of | 11 | jno | of = 0 |

Листинг 10.1. Преобразование регистра символов

```
<1>      :prg_10_1.asm
<2>      model small
<3>      .stack 100h
<4>      .data
<5>      n equ 10          ;количество символов в stroka
<6>      strokadb "acvfgrndup"
<7>      .code
<8>      start:
<9>          mov ax,@data
<10>         mov ds,ax
<11>         xor ax,ax
<12>         mov cx,n
<13>         lea bx,stroka ;адрес stroka в bx
<14>         m1: mov al,[bx] ;очередной символ из stroka в al
<15>         cmp al,61h ;проверить, что код символа не меньше 61h
<16>         jb next ;если меньше, то не обрабатывать
<17>             ;и перейти к следующему символу
<18>         cmp al,7ah ;проверить, что код символа не больше 7ah
<19>         ja next ;если больше, то не обрабатывать
<20>             ;и перейти к следующему символу
<21>         and al,11011111b ;инвертировать 5-й бит
<22>         mov [bx].al ;символ на его место в stroka
```

```

<23>    next:
<24>        inc    bx      ;адресовать следующий символ
<25>        dec    cx      ;уменьшить значение счетчика в cx
<26>        jnz    m1      ;если cx не 0, то переход на m1
<27>    exit:
<28>        mov    ax,4c00h
<29>        int    21h      ;возврат управления операционной системе
<30>    end    start

```

Обратите внимание на строку 25 листинга 10.1. Команда dec уменьшает значение регистра cx на 1. Когда это значение станет равным 0, микропроцессор по результату операции декремента установит флаг zf. Команда в строке 26 анализирует состояние этого флага и, пока он не равен 1 (см. табл. 10.3), передает управление на метку m1. Заметьте, что на место этой команды можно было бы поставить команду jne (см. табл. 10.2). Но для анализа регистра cx в системе команд микропроцессора есть отдельная команда, которую мы сейчас и рассмотрим.

Команды условного перехода и регистр ecx/cx

Архитектура микропроцессора предполагает специфическое использование многих регистров. К примеру, регистр eax/ax/a1 используется как аккумулятор, а регистры bp, sp — для работы со стеком. Регистр ecx/cx тоже имеет определенное функциональное назначение — он выполняет роль счетчика в командах управления циклами и при работе с цепочками символов. Возможно, что функционально команду условного перехода, связанную с регистром ecx/cx, правильнее было бы отнести к этой группе команд.

Синтаксис этой команды условного перехода таков:

jcxz метка_перехода (Jump if cx is Zero) — переход, если cx ноль;
jecxz метка_перехода (Jump Equal ecx Zero) — переход, если ecx ноль.

Эти команды очень удобно использовать при организации цикла и при работе с цепочками символов. На этом уроке мы разберемся со средствами организации циклов в программах на языке ассемблера и покажем работу команды jcxz/jecxz. Следующий урок будет посвящен цепочечным командам. Нужно отметить ограничение, свойственное команде jcxz/jecxz. В отличие от других команд условной передачи управления, команда jcxz/jecxz может адресовать только короткие переходы — на -128 байт или на +127 байт от следующей за ней командой.

Организация циклов

Цикл, как известно, представляет собой важную алгоритмическую структуру, без использования которой не обходится, наверное, ни одна программа. Организовать циклическое выполнение некоторого участка программы можно, к примеру, используя команды условной передачи управления или команду безусловного перехода jmp. Например, подсчитаем количество нулевых байтов в области mas (листинг 10.2).

Листинг 10.2. Подсчет числа нулевых элементов

```
<1> ;prg_10_2.asm
<2> model small
<3> .stack 100h
<4> .data
<5> len equ 10           ;количество элементов в mas
<6> mas db 1,0,9,8,0,7,8,0,2,0
<7> .code
<8> start:
<9>     mov ax,@data
<10>    mov ds,ax
<11>    mov cx,len      ;длину поля mas в cx
<12>    xor ax,ax
<13>    xor si,si
<14> cycl:
<15>    jcxz exit       ;проверка cx на 0, если 0, то выход
<16>    cmp mas[si],0   ;сравнить очередной элемент mas с 0
<17>    jne m1          ;если не равно 0, то на m1
<18>    inc al          ;в al - счетчик нулевых элементов
<19> m1:
<20>    inc si          ;перейти к следующему элементу
<21>    dec cx          ;уменьшить cx на 1
<22>    jmp cycl
<23> exit:
<24>    mov ax,4c00h
<25>    int 21h          ;возврат управления операционной системе
<26> end start
```

Цикл в листинге 10.2 организован тремя командами, `jcxz`, `dec` и `jmp` (строки 15 и 22). Команда `jcxz` выполняет здесь две функции: предотвращает выполнение «пустого» цикла (когда счетчик цикла в `cx` равен нулю) и отслеживает окончание цикла после обработки всех элементов поля `mas`. Команда `dec` после каждой итерации цикла уменьшает значение счетчика в регистре `cx` на 1. Заметьте, что при такой организации цикла все операции по его организации выполняются «вручную». Но, учитывая важность такого алгоритмического элемента, как цикл, разработчики микропроцессора ввели в систему команд группу из трех команд, облегчающую программирование циклов. Эти команды также используют регистр `esx/cx` как счетчик цикла. Дадим краткую характеристику этим командам:

`loop метка_перехода` (`Loop`) — повторить цикл. Команда позволяет организовать циклы, подобные циклам `for` в языках высокого уровня с автоматическим уменьшением счетчика цикла. Работа команды заключается в выполнении следующих действий:

- декремента регистра `esx/cx`;
- сравнения регистра `esx/cx` с нулем;

если $(\text{есх}/\text{cx}) > 0$, то управление передается на метку перехода;

если $(\text{есх}/\text{cx}) = 0$, то управление передается на следующую после `loop` команду.

loop/loopz метка_перехода (Loop till cx <> 0 or Zero Flag = 0) — повторить цикл пока cx <> 0 или zf = 0. Команды loop и loopz — абсолютные синонимы, поэтому используйте ту команду, которая вам больше нравится. Работа команд заключается в выполнении следующих действий:

- декремента регистра ecx/cx;
- сравнения регистра ecx/cx с нулем;
- анализа состояния флага нуля zf;

если (ecx/cx) > 0 и zf = 1, управление передается на метку перехода;
если (ecx/cx) = 0 или zf = 0, управление передается на следующую после loop команду.

loopne/loopnz метка_перехода (Loop till cx <> 0 or nonzero flag=0) — повторить цикл, пока cx <> 0 или zf = 1. Команды loopne и loopnz также абсолютные синонимы. Работа команд заключается в выполнении следующих действий:

- декремента регистра ecx/cx;
 - сравнения регистра ecx/cx с нулем;
 - анализа состояния флага нуля zf;
- если (ecx/cx) > 0 и zf = 0, управление передается на метку перехода;
если (ecx/cx)=0 или zf=1, управление передается на следующую после loop команду.

Команды loop/loopz и loopne/loopnz по принципу своей работы являются взаимообратными. Они расширяют действие команды loop тем, что дополнительно анализируют флаг zf, что дает возможность организовать досрочный выход из цикла, используя этот флаг в качестве индикатора. Типичное использование данных команд связано с операцией поиска определенного значения в последовательности или со сравнением двух чисел.

Недостаток команд организации цикла loop, loop/loopz и loopne/loopnz в том, что они реализуют только короткие переходы (от -128 до +127 байтов). Для работы с длинными циклами придется использовать команды условного перехода и команду jmp (см. листинг 10.2), поэтому постарайтесь освоить оба способа организации циклов. Рассмотрим несколько примеров использования команд loop, loop/loopz и loopne/loopnz для организации циклов.

Программа из листинга 10.2 с использованием команды организации цикла будет выглядеть так, как показано в листинге 10.3.

Программа несложная; интерес в ней представляют строки 20 и 21. Команда loopnz на основании содержимого регистра cx и флага zf принимает решение о продолжении выполнения цикла. Выход из цикла происходит в одном из двух случаев: cx = 0 (просмотрены все элементы поля mas) или zf = 1 (командой cmp обнаружен нулевой элемент). Назначение следующей команды jz (строка 21) в том, чтобы распознать конкретную причину выхода из цикла. Если выход из цикла произо-

щел после просмотра строки, в которой нет нулевых элементов, то `jz` не сработает, и будет выдано сообщение об отсутствии нулевых элементов в строке (строки 7, 23–25). Если выход из цикла произошел в результате обнаружения нулевого элемента, то в регистре `si` будет номер позиции этого элемента в поле `mas`, и при необходимости можно продолжить обработку. В нашем случае мы просто завершаем программу — переходим на метку `exit`.

Листинг 10.3. Подсчет нулевых байтов с использованием команд управления циклом

```
<1>      ;prg_10_3.asm
<2>      model small
<3>      .stack 100h
<4>      .data
<5>      len equ    10          ;количество элементов в mas
<6>      mas db     1,0,9,8,0,7,8,0,2,0
<7>      .code
<8>      start:
<9>          mov  ax,@data
<10>         mov  ds,ax
<11>         mov  cx,len ;длину поля mas в cx
<12>         xor  ax,ax
<13>         xor  si,si
<14>         jcxz exit      ;проверка cx на 0, если 0, то выход
<15>         cycl:
<16>             cmp  mas[si],0  ;сравнить очередной элемент mas с 0
<17>             jne  m1        ;если не равно 0, то на m1
<18>             inc  al        ;в al – счетчик нулевых элементов
<19>         m1:
<20>             inc  si        ;перейти к следующему элементу
<21>             loop cycl
<22>         exit:
<23>             mov  ax,4c00h
<24>             int  21h       ;возврат управления операционной системе
<25>         end   start
```

Заметьте, что у команды `jcxz` в строке 14 осталась только одна функция — не допустить выполнения «пустого» цикла, поэтому несколько изменилось ее место в тексте программы: теперь она стоит перед меткой начала цикла `cycl`. Изменение и контроль содержимого регистра `cx` в процессе выполнения каждой итерации цикла выполняет команда `loop` (строка 21).

Рассмотрим пример, в котором продемонстрируем типичный метод использования команды `loopnz`. Программа из листинга 10.4 ищет первый нулевой элемент в поле `mas`.

Листинг 10.4. Программа prg_10_4.asm

```
<1>      ;prg_10_4.asm
<2>      model small
<3>      .stack 100h
<4>      .data
<5>      len equ    10          ;количество элементов в mas
<6>      mas db     1,0,9,8,0,7,8,0,2,0
```

продолжение ↴

Листинг 10.4 (продолжение)

```
<7>     message    db      "В поле mas нет элементов, равных нулю,$"
<8>     .code
<9>     start:
<10>        mov      ax,@data
<11>        mov      ds,ax
<12>        mov      cx,len ;длину поля mas в cx
<13>        xor      ax,ax
<14>        xor      si,si
<15>        jcxz   exit  ;проверка cx на 0, если 0, то выход
<16>        mov      si,-1 ;готовим si к адресации элементов поля mas
<17>        cyc1:
<18>        inc      si
<19>        cmp      mas[si].0  ;сравнить очередной элемент mas с 0
<20>        loopnz cyc1
<21>        jz       exit  ;выяснение причины выхода из цикла
<22>        ;вывод сообщения, если нет нулевых элементов в mas
<23>        mov      ah,9
<24>        mov      dx,offset message
<25>        int      21h
<26>        exit:
<27>        mov      ax,4c00h
<28>        int      21h  ;возврат управления операционной системе
<29>        end      start
```

Читатели, имеющие даже небольшой опыт программирования на языках высокого уровня, знают, что очень часто возникает необходимость использовать вложенные циклы. Самый простой пример — обработка двухмерного массива. На уроке 12 мы рассмотрим организацию работы с массивами, в том числе и двухмерными. Пока же разберемся с основными принципами организации вложенных циклов. Основная проблема, которая при этом возникает, — как сохранить значения счетчиков в регистре ecx/cx для каждого из циклов. Для временного сохранения счетчика внешнего цикла на время выполнения внутреннего цикла можно использовать несколько способов: задействовать регистры, ячейки памяти или стек. Следующий фрагмент программы содержит три цикла, вложенных один в другой. Этот фрагмент можно рассматривать как шаблон для построения программы с вложенными циклами.

```
<1>     ...
<2>     mov      cx,100      ;количество повторений цикла cyc1_1
<3>     cyc1_1:
<4>     push    cx          ;счетчик цикла cyc1_1 в стек
<5>     ...
<6>     mov      cx,50       ;количество повторений цикла cyc1_2
<7>     cyc1_2:
<8>     push    cx          ;счетчик цикла cyc1_2 в стек
<9>     ...
<10>    mov      cx,25       ;количество повторений цикла cyc1_3
<11>    cyc1_3:
<12>    ...
<13>    loop    cyc1_3      ;команды цикла cyc1_3
```

```

<14>          ...                                ;команды цикла cycl_2
<15>          pop    cx                            ;восстановить счетчик цикла cycl_2
<16>          loop   cycl_2
<17>          ...                                ;команды цикла cycl_1
<18>          pop    cx                            ;восстановить счетчик цикла cycl_2
<19>          loop   cycl_1
<20>          ...

```

В качестве примера рассмотрим фрагмент программы, которая обрабатывает специальным образом некоторую область памяти. Область памяти рассматривается как совокупность пяти полей, содержащих 10 однобайтовых элементов. Требуется заменить все нулевые байты в этой области на значение 0ffh.

Листинг 10.5. Пример использования вложенных циклов

```

<1>      :prg_10_5.asm
<2>      model small
<3>      .stack 100h
<4>      .data
<5>      mas    db    1,0,9,8,0,7,8,0,2,0
<6>              db    1,0,9,8,0,7,8,0,2,0
<7>              db    1,0,9,8,0,7,8,0,2,0
<8>              db    1,0,9,8,0,7,8,0,2,0
<9>              db    1,0,9,8,0,7,8,0,2,0
<10>     .code
<11>     start:
<12>         mov    ax,@data
<13>         mov    ds,ax
<14>         xor    ax,ax
<15>         lea    bx,mas
<16>         mov    cx,5
<17>     cycl_1:
<18>         push   cx
<19>         xor    si,si
<20>         mov    cx,10
<21>     cycl_2:
<22>         cmp    byte ptr [bx+si],0
<23>         jne    no_zero
<24>         mov    byte ptr [bx+si],0ffh
<25>     no_zero:
<26>         inc    si
<27>         loop   cycl_2
<28>         pop    cx
<29>         add    bx,10
<30>         loop   cycl_1
<31>     exit:
<32>         mov    ax,4c00h
<33>         int    21h    ;возврат управления операционной системе
<34>     end   start

```

Комментировать листинг 10.5 нет необходимости — он очень прост для понимания.

Подведем некоторые итоги:

- Язык ассемблера (система команд микропроцессора) имеет достаточно полный и гибкий набор средств организации всевозможных переходов как в пределах текущего сегмента кода, так и во внешние сегменты.
- При организации безусловных переходов возможны переходы как с потерей (`jmp`), так и с запоминанием (`call`) информации о точке передачи управления.
- Принцип работы команд условного перехода основан на том, что микропроцессор по результатам выполнения некоторых команд устанавливает определенные флаги в регистре `eflags/flags`. Команды условного перехода анализируют состояние этих флагов и инициируют передачу управления, исходя из результатов анализа.
- Система команд микропроцессора допускает программирование циклов со счетчиком. Для этого используется регистр `ecx/cx`, в который до входа в цикл должно быть загружено значение счетчика цикла.

11

УРОК

Цепочечные команды

-
- Характеристика средств микропроцессора для обработки цепочек элементов в памяти
 - Операции пересылки и сравнения цепочек
 - Операции для работы с отдельными элементами цепочек
-

На данном уроке будет рассмотрена чрезвычайно интересная группа команд. Понимание принципов работы этих команд и умение грамотно их использовать способны значительно облегчить жизнь программисту, пишущему программы на языке ассемблера. Речь идет о так называемых *цепочечных командах*. Эти команды также называют *командами обработки строк символов*. Названия почти синонимичны. Отличие в том, что под строкой символов здесь понимается последовательность байт, а цепочка — это более общее название для случаев, когда элементы последовательности имеют размер больше байта — слово или двойное слово. Таким образом, цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности элементов следующего размера:

- 8 бит, то есть байт;
- 16 бит, то есть слово;
- 32 бита, то есть двойное слово.

Содержимое этих блоков для микропроцессора не имеет никакого значения. Это могут быть символы, числа и все что угодно. Главное, чтобы размерность элементов совпадала с одной из вышеперечисленных, и эти элементы находились в соседних ячейках памяти.

Всего в системе команд микропроцессора имеются семь операций-примитивов обработки цепочек. Каждая из них реализуется в микропроцессоре тремя командами, в свою очередь, каждая из этих команд работает с соответствующим размером элемента — байтом, словом или двойным словом. Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, осуществляют еще и автоматическое продвижение к следующему элементу данной цепочки.

Перечислим операции-примитивы и команды, с помощью которых они реализуются, а затем подробно их рассмотрим:

- пересылка цепочки:
`movs адрес_приемника,адрес_источника`
`movsb`
`movsw`
`movsd`
- сравнение цепочек:
`cmps адрес_приемника,адрес_источника`
`cmpsb`

`cmpsw`
`cmpsd`

- сканирование цепочки:

`scas адрес_приемника`
`scasb`
`scasw`
`scasd`

- загрузка элемента из цепочки:

`lodс адрес_источника`
`lodsb`
`lodsw`
`lodsd`

- сохранение элемента в цепочке:

`stos адрес_приемника`
`stosb`
`stosw`
`stosd`

- получение элементов цепочки из порта ввода-вывода:

`ins адрес_приемника,номер_порта`
`insb`
`insn`
`insd`

- вывод элементов цепочки в порт ввода-вывода:

`outs номер_порта,адрес_источника`
`outbs`
`outwn`
`outds`

Логически к этим командам нужно отнести и так называемые *префиксы повторения*. Вспомните формат машинной команды и его первые необязательные байты префиксов. Один из возможных типов префиксов — это префиксы повторения. Они предназначены для использования цепочечными командами. Префиксы повторения имеют свои мнемонические обозначения:

`rep`
`repe или repz`
`repne или repnz`

Эти префиксы повторения указываются перед нужной цепочечной командой в поле метки. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполнять в цикле. Отличия приведенных префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: по состоянию регистра `esx/sx` или по флагу нуля `zf`:

префикс повторения `rep` (REPeat). Этот префикс используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек, — соответственно, `movs` и `stos`. Префикс `rep` заставляет данные команды

выполняться, пока содержимое `esx/sx` не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое `esx/sx` на единицу. Та же команда, но без префикса, этого не делает; префиксы повторения `repe` или `repz` (REPeat while Equal or Zero). Эти префиксы являются абсолютными синонимами. Они заставляют цепочечную команду выполнять до тех пор, пока содержимое `esx/sx` не равно нулю или флаг `zf` равен 1. Как только одно из этих условий нарушается, управление передается следующей команде программы. Благодаря возможности анализа флага `zf` наиболее эффективно эти префиксы можно использовать с командами `cmps` и `scas` для поиска отличающихся элементов цепочек;

префиксы повторения `repne` или `repnz` (REPeat while Not Equal or Zero). Эти префиксы также являются абсолютными синонимами. Их действие на цепочечную команду несколько отличается от действий префиксов `repe/repz`. Префиксы `repne/repnz` заставляют цепочечную команду циклически выполнять до тех пор, пока содержимое `esx/sx` не равно нулю или флаг `zf` равен нулю. При невыполнении одного из этих условий работа команды прекращается. Данные префиксы также можно использовать с командами `cmps` и `scas`, но для поиска совпадающих элементов цепочек.

Следующий важный момент, связанный с цепочечными командами, заключается в особенностях формирования физического адреса operandов *адрес_источника* и *адрес_приемника*. Цепочка-источник, адресуемая operandом *адрес_источника*, может находиться в текущем сегменте данных, определяемом регистром `ds`. Цепочка-приемник, адресуемая operandом *адрес_приемника*, должна быть в дополнительном сегменте данных, адресуемом сегментным регистром `es`. Важно отметить, что допускается замена (с помощью префикса замены сегмента) только регистра `ds`, регистр `es` подменять нельзя. Вторые части адресов — смещения цепочек — также находятся в строго определенных местах. Для цепочки-источника это регистр `esi/si` (Source Index register — индексный регистр источника). Для цепочки-получателя это регистр `edi/di` (Destination Index register — индексный регистр приемника). Таким образом, полные физические адреса для operandов цепочечных команд следующие:

- *адрес_источника* — пара `ds:esi/si`;
- *адрес_приемника* — пара `es:edi/di`.

Кстати, вспомните команды `lds` и `les`, которые мы рассматривали на уроке 7. Эти команды позволяют получить полный указатель (сегмент:смещение) на ячейку памяти. Применение их в данном случае очень удобно в силу жесткой регламентации использования регистров для адресации operandов источника и приемника в цепочечных командах.

Вы, наверное, обратили внимание на то, что все семь групп команд, реализующих цепочечные операции-примитивы, имеют похожий по структуре набор команд. В каждом из этих наборов присутствуют одна команда с явным указанием operandов и три команды, не имеющие operandов. На самом деле набор команд микропроцессора имеет соответствующие машинные команды только для цепочечных команд ассемблера без operandов. Команды с operandами транслятор ассемблера использует только для определения типов operandов. После того как выяснен тип элементов цепочек по их описанию в памяти, генерируется одна

из трех машинных команд для каждой из цепочечных операций. По этой причине все регистры, содержащие адреса цепочек, должны быть инициализированы заранее, в том числе и для команд, допускающих явное указание operandов. В силу того, что цепочки адресуются однозначно, нет особого смысла применять команды с operandами. Главное, что вы должны запомнить, — правильная загрузка регистров указателями обязательно требуется до выдачи любой цепочечной команды.

Последний важный момент, касающийся всех цепочечных команд, — это направление обработки цепочки. Есть две возможности:

- от начала цепочки к ее концу, то есть в направлении возрастания адресов;
- от конца цепочки к началу, то есть в направлении убывания адресов.

Как мы увидим ниже, цепочечные команды сами выполняют модификацию регистров, адресующих operandы, обеспечивая тем самым автоматическое продвижение по цепочке. Количество байт, на которые эта модификация осуществляется, определяется кодом команды. А вот знак этой модификации определяется значением *флага направления* df (Direction Flag) в регистре eflags/flags:

- если df = 0, то значения индексных регистров esi/si и edi/di будут автоматически увеличиваться (операция инкремента) цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;
- если df = 1, то значения индексных регистров esi/si и edi/di будут автоматически уменьшаться (операция декремента) цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

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

`cld` (Clear Direction Flag) — очистить флаг направления. Команда сбрасывает флаг направления df в 0.

`std` (Set Direction Flag) — установить флаг направления. Команда устанавливает флаг направления df в 1.

Это вся информация, касающаяся общих свойств цепочечных команд. Далее мы более подробно рассмотрим каждую операцию-примитив и команды, которые ее реализуют. При этом более подробно мы будем рассматривать одну команду в каждой группе цепочечных команд — команду с operandами. Это — более общая команда в смысле ограничений, накладываемых на типы operandов.

Пересылка цепочек

Команды, реализующие эту операцию-примитив, производят копирование элементов из одной области памяти (цепочки) в другую. Размер элемента определяется применяемой командой. Система команд TASM предоставляет программисту четыре команды, работающие с разными размерами элементов цепочки:

`movs` *адрес_приемника, адрес_источника* (MOVe String) — переслать цепочку;

`movsb` (MOVe String Byte) — переслать цепочку байтов;

`movsw` (MOVe String Word) — переслать цепочку слов;

`movsd` (MOVe String Double word) — переслать цепочку двойных слов.

Команда `movs`

`movs адрес_приемника, адрес_источника`

Команда копирует байт, слово или двойное слово из цепочки, адресуемой операндом `адрес_источника`, в цепочку, адресуемую операндом `адрес_приемника`. Размер пересылаемых элементов ассемблер определяет, исходя из атрибутов идентификаторов, указывающих на область памяти приемника и источника. К примеру, если эти идентификаторы были определены директивой `db`, то пересылаться будут байты, если идентификаторы были определены с помощью директивы `dd`, то пересылке подлежат 32-битные элементы, то есть двойные слова. Выше уже было отмечено, что для цепочечных команд с операндами, к которым относится и команда пересылки `movs адрес_приемника, адрес_источника`, не существует машинного аналога. При трансляции, в зависимости от типа операндов, транслятор преобразует ее в одну из трех машинных команд: `movsb`, `movsw` или `movsd`.

Сама по себе команда `movs` пересыпает только один элемент, исходя из его типа, и модифицирует значения регистров `esi/si` и `edi/di`. Если перед командой написать префикс `rep`, то одной командой можно переслать до 64 Кбайт данных (если размер адреса в сегменте 16 бит — `use16`) или до 4 Гбайт данных (если размер адреса в сегменте 32 бита — `use32`). Число пересылаемых элементов должно быть загружено в счетчик — регистры `cx` (`use16`) или `ecx` (`use32`). Перечислим набор действий, которые нужно выполнить в программе для того, чтобы выполнить пересылку последовательности элементов из одной области памяти в другую с помощью команды `movs`. В общем случае этот набор действий можно рассматривать как типовой для выполнения любой цепочечной команды:

1. Установить значение флага `df` в зависимости от того, в каком направлении будут обрабатываться элементы цепочки — в направлении возрастания или убывания адресов.
2. Загрузить указатели на адреса цепочек в памяти в пары регистров `ds:(e)s`i и `es: (e)d`i.
3. Загрузить в регистр `ecx/cx` количество элементов, подлежащих обработке.
4. Выдать команду `movs` с префиксом `rep`.

На примере листинга 11.1 рассмотрим, как эти действия реализуются программно. В этой программе производится пересылка символов из одной строки в другую. Строки находятся в одном сегменте памяти. Для пересылки используется команда-примитив `movs` с префиксом повторения `rep`.

Листинг 11.1. Пересылка строк командой `movs`

```
:prg_11_1.asm
MASM
MODEL    small
STACK    256
```

```

.data
source db "Тестируемая строка",'$' ;строка-источник
dest db 19 DUP (" ") ;строка-приемник
.code
ASSUME assumeds:@data,es:@data
main:
    mov ax,@data
    mov ds,ax
    mov es,ax
    cld
    lea si,source
    lea di,dest
    mov cx,20
repmove dest,source
    lea dx,dest
    mov ah,09h
    int 21h
exit:
    mov ax,4c00h
    int 21h
end main

```

Команды пересылки байтов, слов и двойных слов

Пересылка байтов, слов и двойных слов производится командами `movsb`, `movsw` и `movsd`. Единственной отличительной особенностью этих команд от команды `movs` является то, что последняя может работать с элементами цепочек любого размера — 8, 16 или 32 бита. При трансляции команда `movs` преобразуется в одну из трех команд: `movsb`, `movsw` или `movsd`. Выше мы обсуждали, что решение о том, в какую конкретно команду будет произведено преобразование, принимается транслятором, исходя из размеров элементов цепочек, адреса которых указаны в качестве операндов команды `movs`. Что касается адресов цепочек, то для любой из четырех команд они должны формироваться заранее в регистрах `esi/si` и `edi/di`. Так не проще ли сразу использовать команды пересылки без операндов?

К примеру, посмотрим, как изменится программа из листинга 11.1 при использовании команды `movsb`:

```

...
.data
source db "Пересылаемая строка$" ;строка-источник
dest db 20 DUP (?) ;строка-приемник
.code
ASSUME ASSUME assumeds:@data,es:@data
main:
    cld

```

;сброс флага DF – просмотр строки от начала к концу

продолжение ↗

```
lea    si,source      ;загрузка в ES строки-источника  
lea    di,dest        ;загрузка в DS строки-приемника  
mov    cx,20          ;для префикса гер - длина строки  
 гер movsb             ;пересылка строки
```

Как видим, изменилась только строка с командой пересылки. Отличие в том, что программа из листинга 11.1 может работать с цепочками элементов любой из трех размерностей: 8, 16 или 32 бита, а последний фрагмент — только с цепочками байтов. Далее, как мы и договорились выше, чтобы не загромождать описания, мы будем рассматривать группы команд для операций-примитивов только на примере более общей команды, а вы будете понимать, что на самом деле можно использовать любую из трех команд в соответствующем контексте.

Сравнение цепочек

Команды, реализующие эту операцию-примитив, производят сравнение элементов цепочки-источника с элементами цепочки-приемника. Здесь ситуация с набором команд и методами работы с ними аналогична операции-примитиву пересылки цепочек. TASM предоставляет программисту четыре команды сравнения цепочек, работающие с разными размерами элементов цепочки:

`cmps адрес_приемника, адрес_источника` (CoMPare String) — сравнить строки;
`cmpsb` (CoMPare String Byte) — сравнить строку байт;
`cmpsw` (CoMPare String Word) — сравнить строку слов;
`cmpsd` (CoMPare String Double word) — сравнить строку двойных слов.

Команда `cmps`

Синтаксис команды `cmps`:

`cmps адрес_приемника, адрес_источника`

Здесь:

`адрес_источника` определяет цепочку-источник в сегменте данных. Адрес цепочки должен быть заранее загружен в пару `ds:esi/si`;

`адрес_приемника` определяет цепочку-приемник. Цепочка должна находиться в дополнительном сегменте, и ее адрес должен быть заранее загружен в пару `es:edi/di`.

Алгоритм работы команды `cmps` заключается в последовательном выполнении вычитания (элемент цепочки-источника — элемент цепочки-получателя) над очередными элементами обеих цепочек. Принцип выполнения вычитания командой `cmps` аналогичен команде сравнения `cmp`. Она так же, как и `cmp`, производит вычитание элементов, не записывая при этом результата, и устанавливает флаги `zf`, `sf` и `of`. После выполнения вычитания очередных элементов цепочек командой `cmps` индексные регистры `esi/si` и `edi/di` автоматически изменяются в соответствии со значением флага `df` на значение, равное размеру сравниваемых це-

почек. Чтобы заставить команду `cmps` выполнять несколько раз, то есть производить последовательное сравнение элементов цепочек, необходимо перед командой `cmps` определить префикс повторения. С командой `cmps` можно использовать префиксы повторения `rep/repz` или `repne/repnz`:

○ `rep/repz` — если необходимо организовать сравнение до тех пор, пока не будет выполнено одно из двух условий:

- достигнут конец цепочки (содержимое `esx/sx` равно нулю);
- в цепочках встретились разные элементы (флаг `zf` стал равен нулю).

○ `repne/repnz` — если нужно проводить сравнение до тех пор, пока:

- не будет достигнут конец цепочки (содержимое `esx/sx` равно нулю);
- в цепочках встретились одинаковые элементы (флаг `zf` стал равен единице).

Таким образом, выбрав подходящий префикс, удобно использовать команду `cmps` для поиска одинаковых или различающихся элементов цепочек. Выбор префикса определяется причиной, которая приводит к выходу из цикла. Таких причин может быть две для каждого из префиксов. Для определения конкретной причины наиболее подходящим является способ, использующий команду условного перехода `jcxz`. Ее работа заключается в анализе содержимого регистра `esx/sx`, и если оно равно нулю, то управление передается на метку, указанную в качестве операнда `jcxz`. Так как в регистре `esx/sx` содержится счетчик повторений для цепочечной команды, имеющей любой из префиксов повторения, то, анализируя `esx/sx`, можно определить причину выхода из зацикливания цепочечной команды. Если значение в `esx/sx` не равно нулю, то это означает, что выход произошел по причине совпадения либо несовпадения очередных элементов цепочек. Существует возможность еще больше конкретизировать информацию о причине, приведшей к окончанию операции сравнения. Сделать это можно с помощью команд условной передачи управления (табл. 11.1 и 11.2).

Таблица 11.1. Сочетание команд условной передачи управления с результатами команды `cmps` (для чисел со знаком)

| Причина прекращения операции сравнения | Команда условного перехода, реализующая переход по этой причине |
|--|---|
| операнд_источник > операнд_приемник | <code>jg</code> |
| операнд_источник = операнд_приемник | <code>je</code> |
| операнд_источник <> операнд_приемник | <code>jne</code> |
| операнд_источник < операнд_приемник | <code>jl</code> |
| операнд_источник <= операнд_приемник | <code>jle</code> |
| операнд_источник >= операнд_приемник | <code>jge</code> |

Как определить местоположение очередных совпавших или несовпавших элементов в цепочках? Вспомните, что после каждой итерации цепочечная команда автоматически осуществляет инкремент/декремент значения адреса в соответствующих индексных регистрах. Поэтому после выхода из цикла в этих регистрах будут находиться адреса элементов, находящихся в цепочке после (!) элементов,

которые послужили причиной выхода из цикла. Для получения истинного адреса этих элементов необходимо скорректировать содержимое индексных регистров, увеличив либо уменьшив значение в них на длину элемента цепочки.

Таблица 11.2. Сочетание команд условной передачи управления с результатами команды cmps (для чисел без знака)

| Причина прекращения операции сравнения | Команда условного перехода, реализующая переход по этой причине |
|--|---|
| операнд_источник > операнд_приемник | ja |
| операнд_источник = операнд_приемник | je |
| операнд_источник <> операнд_приемник | jne |
| операнд_источник < операнд_приемник | jb |
| операнд_источник <= операнд_приемник | jbe |
| операнд_источник >= операнд_приемник | jae |

В качестве примера рассмотрим программу из листинга 11.2, которая сравнивает две строки, находящиеся в одном сегменте. Используется команда cmps. Префикс повторения — repe.

Листинг 11.2. Сравнение двух строк командой cmps

```
<1>      ;prg_11_2.asm
<2>      MODEL small
<3>      STACK 256
<4>      .data
<5>      match db    0ah,0dh,'Строки совпадают.', '$'
<6>      faileddb db    0ah,0dh,'Строки не совпадают', '$'
<7>      string1 db    "0123456789",0ah,0dh,'$';исследуемые строки
<8>      string2 db    "0123406789", '$'
<9>      .code
<10>     ASSUME ds:@data,es:@data ;привязка DS и ES к сегменту данных
<11>     main:
<12>         mov    ax,@data    ;загрузка сегментных регистров
<13>         mov    ds,ax
<14>         mov    es,ax ;настройка ES на DS
<15>         ;вывод на экран исходных строк string1 и string2
<16>         mov    ah,09h
<17>         lea    dx,string1
<18>         int    21h
<19>         lea    dx,string2
<20>         int    21h
<21>         ;сброс флага DF – сравнение в направлении возрастания адресов
<22>         cld
<23>         lea    si,string1 ;загрузка в si смещения string1
<24>         lea    di,string2 ;загрузка в di смещения string2
<25>         mov    cx,10 ;длина строки для префикса repe
<26>         ;сравнение строк (пока сравниваемые элементы строк равны)
<27>         ;выход при обнаружении несовпадшего элемента
<28>         cyc1:
<29>             repe  cmps  string1,string2
<30>             jcxz equal ;cx=0, то есть строки совпадают
<31>             jne   not_match ;если не равны – переход на not_match
```

```

<32>    equal:           ;иначе, если совпадают, то
<33>        mov ah,09h ;вывод сообщения
<34>        lea dx,match
<35>        int 21h
<36>        jmp exit      ;выход
<37>    not_match:       ;не совпали
<38>        mov ah,09h
<39>        lea dx,failed
<40>        int 21h ;вывод сообщения
<41> ;теперь, чтобы обработать несовпавший элемент в строке, необходимо
<42> ;уменьшить значения регистров si и di
<43>        dec si
<44>        dec di
<45> ;сейчас в ds:si и es:di адреса несовпавших элементов
<46> ;здесь вставить код по обработке несовпавшего элемента
<47> ;после этого продолжить поиск в строке:
<48>        inc si
<49>        inc di
<50>        jmp cyc1
<51>    exit:            ;выход
<52>        mov ax,4c00h
<53>        int 21h
<54>    end main         ;конец программы

```

Программа достаточно прозрачна, только два момента, на мой взгляд, требуют пояснения. Это, во-первых, строки 43 и 44, в которых мы скорректировали адреса очередных элементов для получения адресов несовпавших элементов. Вы должны понимать, что если сравниваются цепочки с элементами слов или двойных слов, то корректировать содержимое esi/si и edi/di нужно на 2 и 4 байта, соответственно. И во-вторых, строки 48–50. Смысл их в том, что для просмотра оставшейся части строк необходимо установить указатели на следующие элементы строк за последними несовпавшими. После этого можно повторить весь процесс просмотра и обработки несовпавших элементов в оставшихся частях строк.

Команды сравнения байтов, слов и двойных слов

Аналогично ситуации с набором команд для пересылки цепочки в группе команд сравнения есть отдельные команды сравнения цепочек байт, слов, двойных слов — `cmpsb`, `cmpsw` и `cmpsd`, соответственно. Для этих команд все рассуждения аналогичны тем, что были приведены при обсуждении команд пересылки. Ассемблер преобразует команду `cmps` в одну из машинных команд `cmpsb`, `cmpsw` или `cmpsd`, в зависимости от размера элементов сравниваемых цепочек.

Сканирование цепочек

Команды, реализующие эту операцию-примитив, производят поиск некоторого значения в области памяти. Логически эта область памяти рассматривается как последовательность (цепочка) элементов фиксированной длины размером 8, 16

или 32 бита. Искомое значение предварительно должно быть помещено в регистр `a1/ax/eax`. Выбор конкретного регистра из этих трех должен быть согласован с размером элементов цепочки, в которой осуществляется поиск. Система команд микропроцессора предоставляет программисту четыре команды сканирования цепочки. Выбор конкретной команды определяется размером элемента:

`scas` *адрес_приемника* (SCAning String) — сканировать цепочку;
`scasb` (SCAning String Byte) — сканировать цепочку байт;
`scasw` (SCAning String Word) — сканировать цепочку слов;
`scasd` (SCAning String Double Word) — сканировать цепочку двойных слов.

Команда scas

`scas` *адрес_приемника*

Команда имеет один операнд, обозначающий местонахождение цепочки в дополнительном сегменте (адрес цепочки должен быть заранее сформирован в `es:edi/di`). Транслятор анализирует тип идентификатора *адрес_приемника*, который обозначает цепочку в сегменте данных, и формирует одну из трех машинных команд, `scasb`, `scasw` или `scasd`. Условие поиска для каждой из этих трех команд находится в строго определенном месте. Так, если цепочка описана с помощью директивы `db`, то искомый элемент должен быть байтом и находиться в `a1`, а сканирование цепочки осуществляется командой `scasb`; если цепочка описана с помощью директивы `dw`, то это — слово в `ax`, и поиск ведется командой `scasw`; если цепочка описана с помощью директивы `dd`, то это двойное слово в `eax`, и поиск ведется командой `scasd`. Принцип поиска тот же, что и в команде сравнения `cmps`, то есть последовательное выполнение вычитания (*содержимое_регистра_аккумулятора – содержимое_очередного_элемента_цепочки*). В зависимости от результатов вычитания производится установка флагов, при этом сами операнды не изменяются. Так же как и в случае команды `cmps`, с командой `scas` удобно использовать префиксы `repe/repz` или `repne/repnz`:

- `repe` или `repz` — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
 - достигнут конец цепочки (содержимое `esx/sx` равно 0);
 - в цепочке встретился элемент, отличный от элемента в регистре `a1/ax/eax`.
- `repne` или `repnz` — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
 - достигнут конец цепочки (содержимое `esx/sx` равно 0);
 - в цепочке встретился элемент, совпадающий с элементом в регистре `a1/ax/eax`.

Таким образом, команда `scas` с префиксом `repe/repz` позволяет найти элемент цепочки, отличающийся по значению от заданного в аккумуляторе. Команда `scas` с префиксом `repne/repnz` позволяет найти элемент цепочки, совпадающий по значению с элементом в аккумуляторе. В качестве примера рассмотрим листинг 11.3, который производит поиск символа в строке. В программе использу-

ется команда-примитив scas. Символ задается явно (строка 20). Префикс повторения — repne.

Листинг 11.3. Поиск символа в строке командой scas

```
<1>      ;prg_11_3.asm
<2>      MASM
<3>      MODEL small
<4>      STACK 256
<5>      .data
<6>      ;тексты сообщений
<7>      fnd    db    0ah,0dh,'Символ найден! ','$'
<8>      nochar db    0ah,0dh,'Символ не найден. ','$'
<9>      ;строка для поиска
<10>     stringdb   "Поиск символа в этой строке.",0ah,0dh,'$'
<11>     .code
<12>     ASSUME ds:@data,es:@data
<13>     main:
<14>         mov    ax,@data
<15>         mov    ds,ax
<16>         mov    es,ax      ;настройка ES на DS
<17>         mov    ah,09h
<18>         lea    dx,string
<19>         int    21h      ;вывод сообщения string
<20>         mov    al,'а' ;символ для поиска – "а"(кириллица)
<21>         cld    ;сброс флага df
<22>         lea    di,string  ;загрузка в es:di смещения строки
<23>         mov    cx,29      ;для префикса repne – длина строки
<24>     ;поиск в строке (пока искомый символ и символ в строке не совпадут)
<25>     ;выход при первом совпадении
<26>     repne scas   string
<27>         je    found  ;если равны – переход на обработку,
<28>     failed:           ;иначе выполняем некоторые действия
<29>     ;вывод сообщения о том, что символ не найден
<30>         mov    ah,09h
<31>         lea    dx,nochar
<32>         int    21h      ;вывод сообщения nochar
<33>         jmp    exit   ;на выход
<34>     found:           ;совпали
<35>         mov    ah,09h
<36>         lea    dx,fnd
<37>         int    21h      ;вывод сообщения fnd
<38>     ;теперь, чтобы узнать место, где совпал элемент в строке,
<39>     ;необходимо уменьшить значение в регистре di и вставить нужный обработчик
<40>     ;      dec di
<41>     ... вставьте обработчик
<42>     exit:             ;выход
<43>         mov    ax,4c00h
<44>         int    21h
<45>     end   main
```

Сканирование строки байтов, слов, двойных слов

Система команд микропроцессора, так же, как и в случае операций-примитивов пересылки и сравнения, предоставляет вам команды сканирования, явно указывающие размер элемента цепочки — scasb, scasw или scasd. Помните, что даже если вы этого не делаете, то ассемблер все равно преобразует команду scas в одну из этих трех машинных команд.

Загрузка элемента цепочки в аккумулятор

Эта операция-примитив позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор al, ah или eax. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения). Возможный размер извлекаемого элемента определяется применяемой командой. Программист может использовать четыре команды загрузки элемента цепочки в аккумулятор, работающие с элементами разного размера:

`lods адрес_источника (LOaD String)` — загрузить элемент из цепочки в регистр-аккумулятор al/ah/eax.

`lodsb (LOaD String Byte)` — загрузить байт из цепочки в регистр al.

`lodsw (LOaD String Word)` — загрузить слово из цепочки в регистр ah.

`lodsd (LOaD String Double Word)` — загрузить двойное слово из цепочки в регистр eax.

Рассмотрим работу этих команд на примере lods.

Команда lods

`lods адрес_источника (LOaD String)` — загрузить элемент из цепочки в аккумулятор al/ah/eax.

Команда имеет один operand, обозначающий строку в основном сегменте данных. Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому пары регистров ds:esi/si, и поместить его в регистр eax/ah/al. При этом содержимое esi/si подвергается инкременту или декременту (в зависимости от состояния флага df) на значение, равное размеру элемента. Эту команду удобно использовать после команды scas, локализующей местоположение искомого элемента в цепочке. Префикс повторения в этой команде может и не понадобиться — все зависит от логики программы.

В качестве примера рассмотрим листинг 11.4. Программа сравнивает командой `cmps` две цепочки байт в памяти `string1` и `string2` и помещает первый несовпадший байт из `string2` в регистр al. Для загрузки этого байта в регистр-аккумулятор al используется команда `lods`. Префикса повторения в команде `lods` нет, так как он попросту не нужен.

Листинг 11.4. Использование lods для загрузки байта в регистр al

```
<1> :prg_11_4.asm
<2> MASM
<3> MODEL small
<4> STACK 256
<5> .data
<6> ;строки для сравнения
<7> string1    db      "Поиск символа в этой строке.",0ah,0dh,'$'
<8> string2    db      "Поиск символа не в этой строке.",0ah,0dh,'$'
<9> mes_eqdb   "Строки совпадают.",0ah,0dh,'$'
<10> fnd        db      "Несовпадший элемент в регистре al",0ah,0dh,'$'
<11> .code
<12> ;привязка ds и es к сегменту данных
<13> assume ds:@data,es:@data
<14> main:
<15>     mov ax,@data      ;загрузка сегментных регистров
<16>     mov ds,ax
<17>     mov es,ax      ;настройка es на ds
<18>     mov ah,09h
<19>     lea dx,string1
<20>     int 21h          ;вывод string1
<21>     lea dx,string2
<22>     int 21h          ;вывод string2
<23>     cld             ;сброс флага df
<24>     lea di,string1
<25>     lea si,string2
<26>     mov cx,29         ;для префикса repe – длина строки
<27>                 ;поиск в строке (пока нужный символ и символ
<28>                 ;в строке не равны)
<29>     ;выход – при первом несовпадшем
<30>     repe cmps string1,string2
<31>     jcxz eq1          ;если равны – переход на eq1
<32>     jmp no_eq        ;если не равны – переход на no_eq
<33>     eq1:               ;выводим сообщение о совпадении строк
<34>     mov ah,09h
<35>     lea dx,mes_eq
<36>     int 21h          ;вывод сообщения mes_eq
<37>     jmp exit          ;на выход
<38>     no_eq:            ;обработка несовпадения элементов
<39>     mov ah,09h
<40>     lea dx,fnd
<41>     int 21h          ;вывод сообщения fnd
<42>     ;теперь, чтобы извлечь несовпадший элемент из строки
<43>     ;в регистр-аккумулятор,
<44>     ;уменьшаем значение регистра si и тем самым перемещаемся
<45>     ;к действительной позиции элемента в строке
<46>     dec si            ;команда lods использует ds:si-адресацию
<47>     ;теперь ds:si указывает на позицию в string2
<48>     lods string2      ;загрузим элемент из строки в AL
<49>     ;нетрудно догадаться, что в нашем примере это символ – "H"
```

```

<49>    exit:           ;выход
<50>    mov    ax,4c00h
<51>    int    21h
<52>    end    main

```

Загрузка в регистр al/ax/eax байтов, слов, двойных слов

Команды загрузки байта в регистр al (`lodsb`), слова в регистр ax (`lodsw`), двойного слова в регистр eax (`lodsd`) аналогичны другим цепочечным командам. Они являются вариантами команды `lods`. Каждая из этих команд работает с цепочками из элементов определенного размера. Предварительно вы должны загрузить длину цепочки и ее адрес в регистры ecx/cx и ds:esi/si.

Перенос элемента из аккумулятора в цепочку

Эта операция-примитив позволяет произвести действие, обратное команде `lods`, то есть сохранить значение из регистра-аккумулятора в элементе цепочки. Эту операцию удобно использовать вместе с операциями поиска (сканирования) `scans` и загрузки `lods` с тем, чтобы, найдя нужный элемент, извлечь его в регистр и записать на его место новое значение. Команды, поддерживающие эту операцию-примитив, могут работать с элементами размером 8, 16 или 32 бита. TASM предоставляет программисту четыре команды сохранения элемента цепочки из регистра-аккумулятора, работающие с элементами разного размера:

`stos адрес_приемника` (STOrе String) — сохранить элемент из регистра-аккумулятора al/ax/eax в цепочке;

`stosb` (STOrе String Byte) — сохранить байт из регистра al в цепочке;

`stosw` (STOrе String Word) — сохранить слово из регистра ax в цепочке;

`stosd` (STOrе String Double Word) — сохранить двойное слово из регистра eax в цепочке.

Команда `stos`

`stos адрес_приемника` (STOrе String) — сохранить элемент из регистра-аккумулятора al/ax/eax в цепочке.

Команда имеет один operand `адрес_приемника`, адресующий цепочку в дополнительном сегменте данных. Работа команды заключается в том, что она пересыпает элемент из аккумулятора (регистра eax/ax/al) в элемент цепочки по адресу, соответствующему содержимому пары регистров es:edi/di. При этом содержимое edi/di подвергается инкременту или декременту (в зависимости от состояния флага df) на значение, равное размеру элемента цепочки.

Префикс повторения в этой команде может и не понадобиться — все зависит от логики программы. Например, если использовать префикс повторения `rep`, то можно применить команду для инициализации области памяти некоторым фиксированным значением.

В качестве примера рассмотрим листинг 11.5. Программа производит замену в строке всех символов «`a`» на другой символ. Символ для замены вводится с клавиатуры.

Листинг 11.5. Замена командой `stos` символа в строке на введенный с клавиатуры

```
:prg_11_5.asm
MASM
MODEL    small
STACK    256
.data
;сообщения
fnddb    0ah,0dh,'Символ найден','$'
nochar   db     0ah,0dh,'Символ не найден.', '$'
mes1     db     0ah,0dh,'Исходная строка: ','$'
string   db     "Поиск символа в этой строке.",0ah,0dh,'$' ;строка для поиска
mes2     db     0ah,0dh,'Введите символ, на который следует заменить найденный'
db      0ah,0dh,'$'
mes3     db     0ah,0dh,'Новая строка: ","$'
.code
    assumeds:@data,es:@data ;привязка ds и es
                           ;к сегменту данных
main:
    mov ax,@data           ;точка входа в программу
                           ;загрузка сегментных регистров
    mov ds,ax
    mov es,ax               ;настройка es на ds
    mov ah,09h
    lea dx,mes1
    int 21h                 ;вывод сообщения mes1
    lea dx,string
    int 21h                 ;вывод string
    mov al,'a'              ;символ для поиска — "а"(кириллица)
    cld                     ;сброс флага df
    lea di,string
    mov di,string            ;загрузка в di смещения string
    mov cx,29                ;для префикса repne — длина строки
;поиск в строке string до тех пор, пока
;символ в al и очередной символ в строке
;не равны: выход — при первом совпадении
cycl:
    repne scas string
    je found                ;если элемент найден, то переход на found
failed:
    mov ah,09h
    lea dx,nochar
    int 21h
    jmp exit                ;переход на выход
```

```

found:
    mov    ah,09h
    lea    dx,fnd
    int    21h ;вывод сообщения об обнаружении символа
;корректируем di для получения значения
;действительной позиции совпавшего элемента
;в строке и регистре al
    dec    di
new_char:   ;блок замены символа
    mov    ah,09h
    lea    dx,mes2
    int    21h ;вывод сообщения mes2
;ввод символа с клавиатуры
    mov    ah,01h
    int    21h ;в al – введенный символ
    stos  string;сохраним введенный символ
              ;(из al) в строке
              ;string в позиции старого символа
    mov    ah,09h
    lea    dx,mes3
    int    21h ;вывод сообщения mes3
    lea    dx,string
    int    21h ;вывод сообщения string
;переход на поиск следующего символа "a" в строке
    inc    di ;указатель в строке string на символ, следующий после совпавшего,
    jmp    cycl ;на продолжение просмотра string
exit:      ;выход
    mov    ax,4c00h
    int    21h
end main   ;конец программы

```

Сохранение в цепочке байта, слова, двойного слова из регистра al/ax/eax

Команды `stosb`, `stosw` и `stosd`, аналогично другим цепочечным операциям, являются вариантами команды `stos`. Каждая из этих команд работает с цепочками из элементов определенного размера. Предварительно вы должны загрузить длину цепочки и ее адрес в регистры `esch/cx` и `es:edi/di`.

Следующие две команды появились впервые в системе команд микропроцессора i386. Они позволяют организовать эффективную передачу данных между портами ввода-вывода и цепочками в памяти. Следует отметить, что эти две команды позволяют достичь более высокой скорости передачи данных по сравнению с той скоростью, которую может обеспечить контроллер DMA (Direct Memory Access — прямой доступ к памяти). Контроллер DMA — это специальная микросхема, предназначенная для того, чтобы освободить микропроцессор от управления вводом-выводом больших массивов данных между внешним устройством (диском) и памятью.

Ввод элемента цепочки из порта ввода-вывода

Данная операция позволяет произвести ввод цепочки элементов из порта ввода-вывода и реализуется командой `ins`, имеющей следующий формат:

`ins адрес_приемника, номер_порта` (Input String) — ввести элементы из порта ввода-вывода в цепочку.

Эта команда вводит элемент из порта, номер которого находится в регистре `dx`, в элемент цепочки, адрес которого определяется операндом `адрес_приемника`. Несмотря на то что цепочка, в которую вводится элемент, адресуется указанием этого операнда, ее адрес должен быть явно сформирован в паре регистров `es:edi/di`. Размер элементов цепочки должен быть согласован с размерностью порта; он определяется директивой резервирования памяти, с помощью которой выделяется память для размещения элементов цепочки. После пересылки команда `ins` производит коррекцию содержимого `edi/di` на величину, равную размеру элемента, участвовавшего в операции пересылки. Как обычно, при работе цепочечных команд учитывается состояние флага `df`.

Подобно командам, реализующим рассмотренные выше цепочечные операции-примитивы, транслятор преобразует команду `ins` в одну из трех машинных команд без операндов, работающих с цепочками элементов определенного размера:

`insb` (INput String Byte) — ввести из порта цепочку байтов;

`insw` (INput String Word) — ввести из порта цепочку слов;

`insd` (INput String Double Word) — ввести из порта цепочку двойных слов.

К примеру, выведем 10 байтов из области памяти `pole` в порт `5000h`.

```
.data
pole    db      10 dup (" ")
.code
...
push  ds
pop   es      ;настройка es на ds
mov   dx,5000h
lea   di,pole
mov   cx,10
repinsb
...
```

Вывод элемента цепочки в порт ввода-вывода

Данная операция позволяет произвести вывод элементов цепочки в порт ввода-вывода. Она реализуется командой `outs`, имеющей следующий формат:

`outs номер_порта, адрес_источника` (Output String) — вывести элементы из цепочки в порт ввода-вывода.

Эта команда выводит элемент цепочки в порт, номер которого находится в регистре dx. Адрес элемента цепочки определяется операндом *адрес_источника*. Несмотря на то что цепочка, из которой выводится элемент, адресуется указанием этого операнда, значение адреса должно быть явно сформировано в паре регистров ds:esi/si. Размер структурных элементов цепочки должен быть согласован с размерностью порта. Он определяется директивой резервирования памяти, с помощью которой выделяется память для размещения элементов цепочки. После пересылки команда outs производит коррекцию содержимого esi/si на величину, равную размеру элемента цепочки, участвовавшего в операции пересылки. При этом, как обычно, учитывается состояние флага df.

Подобно команде ins транслятор преобразует команду outs в одну из трех машинных команд без операндов, работающих с цепочками элементов определенного размера:

outsb (OUTput String Byte) – вывести цепочку байтов в порт ввода-вывода;
outsw (OUTtput String Word) – вывести цепочку слов в порт ввода-вывода;
outsd (OUTput String Double Word) – вывести цепочку двойных слов в порт ввода-вывода.

В качестве примера рассмотрим фрагмент программы, которая выводит последовательность символов в порт ввода-вывода, соответствующего принтеру (номер 378 (lpt1)).

```
.data
str_pech db      "Текст для печати"
.code
...
    mov    dx,378h
    lea    di,str_pech
    mov    cx,16
repoutsb
...
```

В заключение напомню, что для организации работы с портами недостаточно знать их номера и назначение. Не менее важно знать и понимать алгоритмы их работы. Эти сведения можно найти в документации на устройство (но, к сожалению, далеко не всегда).

Подведем некоторые итоги:

- Система команд микропроцессора имеет очень интересную группу команд, позволяющих производить действия над блоками элементов до 64 Кбайт или 4 Гбайт, в зависимости от установленной разрядности адреса use16 или use32.
- Эти блоки логически могут представлять собой последовательности элементов с любыми значениями, хранящимися в памяти в виде двоичных кодов. Единственное ограничение состоит в том, что размеры элементов этих блоков памяти имеют фиксированный размер 8, 16 или 32 бита.

- Команды обработки строк предоставляют возможность выполнения семи операций-примитивов, обрабатывающих цепочки поэлементно.
- Каждая операция-примитив представлена тремя разными машинными командами и одной псевдокомандой, которая преобразуется транслятором в одну из трех вышеупомянутых машинных команд. Это преобразование происходит в зависимости от типа указанных в ней операндов.
- Микропроцессор всегда предполагает, что строка-приемник находится в дополнительном сегменте (адресуемом посредством сегментного регистра es), а строка-источник — в сегменте данных (адресуемом посредством сегментного регистра ds).
- Микропроцессор адресует строку-приемник через регистр edi/di, а строку-источник — через регистр esi/si.
- Допускается переопределять сегмент для строки-источника, для строки-приемника этого делать нельзя.
- Особенность работы цепочечных команд состоит в том, что они автоматически выполняют приращение или уменьшение содержимого регистров edi/di и esi/si в зависимости от используемой цепочечной команды. Что именно происходит с этими регистрами, определяется состоянием флага df, которым управляют команды cld и std. Значение, на которое изменяется содержимое индексных регистров, определяется типом элементов строки или кодом операции цепочечной команды.

12

УРОК

Сложные структуры данных

-
- Понятие сложного типа данных в ассемблере
 - Средства ассемблера для создания и обработки сложных структур данных
 - Массивы
 - Структуры
 - Объединения
 - Записи
-

На предыдущих уроках при разработке программ мы использовали данные двух типов:

- Непосредственные данные, представляющие собой числовые или символьные значения, являющиеся частью команды. Непосредственные данные формируются программистом в процессе написания программы для конкретной команды ассемблера.
- Данные, описываемые с помощью ограниченного набора директив резервирования памяти, позволяющих выполнить самые элементарные операции по размещению и инициализации числовой и символьной информации. При обработке этих директив ассемблер сохраняет в своей таблице символов информацию о местоположении данных (значения сегментной составляющей адреса и смещения) и типе данных, то есть единицах памяти, выделяемых для размещения данных в соответствии с директивой резервирования и инициализации данных.

Эти два типа данных являются *элементарными*, или *базовыми*; работа с ними поддерживается на уровне системы команд микропроцессора. Используя данные этих типов, можно формализовать и запрограммировать практически любую задачу. Но насколько это будет удобно — вот вопрос.

Обработка информации, в общем случае, процесс очень сложный. Это косвенно подтверждает популярность языков высокого уровня. Одно из несомненных достоинств языков высокого уровня — поддержка развитых структур данных. При их использовании программист освобождается от решения конкретных проблем, связанных с представлением числовых или символьных данных, и получает возможность оперировать информацией, структура которой в большей степени отражает особенности предметной области решаемой задачи. В то же самое время, чем выше уровень такой абстракции данных от конкретного их представления в компьютере, тем большая нагрузка ложится на компилятор с целью создания действительно эффективного кода. Ведь нам уже известно, что в конечном итоге все, написанное на языке высокого уровня, в компьютере будет представлено на уровне машинных команд, работающих только с базовыми типами данных. Таким образом, самая эффективная программа — программа, написанная в машинных кодах, но писать сегодня большую программу в машинных кодах — занятие, не имеющее слишком большого смысла.

С целью облегчения разработки программ в язык ассемблера была введена возможность использования нескольких сложных типов данных. Они строятся на

основе базовых типов данных, которые являются как бы кирпичиками для их построения. Введение сложных типов данных позволяет несколько сгладить различия между языками высокого уровня и ассемблером. У программиста появляется возможность сочетания преимуществ языка ассемблера и языков высокого уровня (в направлении абстракции данных), что в итоге повышает эффективность конечной программы.

TASM поддерживает следующие сложные типы данных:

- массивы;
- структуры;
- объединения;
- записи.

Разберемся более подробно с тем, как определить данные этих типов в программе и организовать работу с ними.

Массивы

Дадим формальное определение: *массив* – структурированный тип данных, состоящий из некоторого числа элементов одного типа.

Для того чтобы разобраться в возможностях и особенностях обработки массивов в программах на ассемблере, нужно ответить на следующие вопросы:

- Как описать массив в программе?
- Как инициализировать массив, то есть как задать начальные значения его элементов?
- Как организовать доступ к элементам массива?
- Как организовать выполнение типовых операций с массивами?

Описание и инициализация массива в программе

Специальных средств описания массивов в программах ассемблера, конечно, нет. При необходимости использовать массив в программе его нужно моделировать одним из следующих способов:

- перечислением элементов массива в поле operandов одной из директив описания данных. При перечислении элементы разделяются запятыми. Например:
; массив из 5 элементов. Размер каждого элемента 4 байта:
mas dd 1,2,3,4,5

- используя оператор повторения dup. К примеру:

; массив из 5 нулевых элементов. Размер каждого элемента 2 байта:
mas dw 5 dup (0)

Такой способ определения используется для резервирования памяти с целью размещения и инициализации элементов массива;

- используя директивы `label` и `rept`. Пара этих директив может облегчить описание больших массивов в памяти и повысить наглядность такого описания. Директива `rept` относится к макросредствам языка ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой `endm`. К примеру, определим массив байт в области памяти, обозначенной идентификатором `mas_b`. В данном случае директива `label` определяет символьическое имя `mas_b` аналогично тому, как это делают директивы резервирования и инициализации памяти. Достоинство директивы `label` в том, что она не резервирует память, а лишь определяет характеристики объекта. В данном случае объект — это ячейка памяти. Используя несколько директив `label`, записанных одна за другой, можно присвоить одной и той же области памяти разные имена и типы, что и сделано в следующем фрагменте:

```
...
n=0
...
```

```
mas_b label byte
mas_w label word
rept 4
    dw    0f1f0h
endm
```

В результате в памяти будет создана последовательность из четырех слов `f1f0`. Эту последовательность можно трактовать как массив байт или слов в зависимости от того, какое имя области мы будем использовать в программе — `mas_b` или `mas_w`;

- использованием цикла для инициализации значениями области памяти, которую можно будет впоследствии трактовать как массив. Посмотрим на примере листинга 12.1, каким образом это делается.

Листинг 12.1. Инициализация массива в цикле

```
:prg_12_1.asm
MASM
MODEL small
STACK 256
.data
mes db 0ah,0dh,'Массив - "','"$'
mas db 10 dup (?)           ;исходный массив
i   db 0
.code
main:
        mov ax,@data
        mov ds,ax
        xor ax,ax          ;обнуление ax
        mov cx,10            ;значение счетчика цикла в cx
        mov si,0              ;индекс начального элемента в cx
go:                                ;цикл инициализации
        mov bh,i             ;i в bh
        mov mas[si],bh       ;запись в массив i
        inc i                ;инкремент i
        inc si               ;продвижение к следующему элементу массива
```

```

loop go          ;повторить цикл
;вывод на экран получившегося массива
    mov cx,10
    mov si,0
    mov ah,09h
    lea dx,mes
    int 21h
show:
    mov ah,02h      ;функция вывода значения из al на экран
    mov dl,mas[si]
    add dl,30h      ;преобразование числа в символ
    int 21h
    inc si
    loop show
exit:
    mov ax,4c00h    ;стандартный выход
    int 21h
end main        ;конец программы

```

Доступ к элементам массива

При работе с массивами необходимо четко представлять себе, что все элементы массива располагаются в памяти компьютера последовательно. Само по себе такое расположение ничего не говорит о назначении и порядке использования этих элементов. И только лишь программист с помощью составленного им алгоритма обработки определяет то, как нужно трактовать эту последовательность байт, составляющих массив. Так, одну и ту же область памяти можно трактовать как одномерный массив, и одновременно те же самые данные могут трактоваться как двумерный массив. Все зависит только от алгоритма обработки этих данных в конкретной программе. Сами по себе данные не несут никакой информации о своем «смысловом», или логическом типе. Помните об этом принципиальном моменте.

Эти же соображения можно распространить и на индексы элементов массива. Ассемблер не подозревает об их существовании и ему абсолютно все равно, каковы их численные смысловые значения. Для того чтобы локализовать определенный элемент массива, к его имени нужно добавить индекс. Так как мы моделируем массив, то должны позаботиться и о моделировании индекса. В языке ассемблера индексы массивов — это обычные адреса, но с ними работают особым образом. Другими словами, когда при программировании на ассемблере мы говорим об индексе, то, скорее, подразумеваем под этим не номер элемента в массиве, а некоторый адрес. Давайте еще раз обратимся к описанию массива. К примеру, в программе статически определена последовательность данных:

masdw 0,1,2,3,4,5

Пусть эта последовательность чисел трактуется как одномерный массив. Размерность каждого элемента определяется директивой dw, то есть она равна двум

байтам. Чтобы получить доступ к третьему элементу, нужно к адресу массива прибавить 6. Нумерация элементов массива в ассемблере начинается с нуля. То есть в нашем случае речь, фактически, идет о 4-м элементе массива — 3, но об этом знает только программист; микропроцессору в данном случае все равно — ему нужен только адрес. В общем случае для получения адреса элемента в массиве необходимо начальный (базовый) адрес массива сложить с произведением индекса (номер элемента минус единица) этого элемента на размер элемента массива:

база + (индекс*размер элемента)

Архитектура микропроцессора предоставляет достаточно удобные программно-аппаратные средства для работы с массивами. К ним относятся базовые и индексные регистры, позволяющие реализовать несколько режимов адресации данных. Используя данные режимы адресации, можно организовать эффективную работу с массивами в памяти. Вспомним эти режимы:

○ индексная адресация со смещением — режим адресации, при котором эффективный адрес формируется из двух компонентов:

- постоянного (базового) — указанием прямого адреса массива в виде имени идентификатора, обозначающего начало массива;
- переменного (индексного) — указанием имени индексного регистра.

К примеру:

```
mas dw 0,1,2,3,4,5
```

```
...
```

```
    mov si,4
```

; поместить 3-й элемент массива mas в регистр ax:

```
    mov ax,mas[si]
```

○ базовая индексная адресация со смещением — режим адресации, при котором эффективный адрес формируется максимум из трех компонентов:

- постоянного (необязательного компонента), в качестве которого может выступать прямой адрес массива в виде имени идентификатора, обозначающего начало массива, или непосредственное значение;
- переменного (базового) — указанием имени базового регистра;
- переменного (индексного) — указанием имени индексного регистра.

Этот вид адресации удобно использовать при обработке двухмерных массивов. Пример использования этой адресации мы рассмотрим ниже при изучении особенностей работы с двухмерными массивами.

Напомним, что в качестве базового регистра может использоваться любой из восьми регистров общего назначения. В качестве индексного регистра также можно использовать любой регистр общего назначения, за исключением esp/sp.

Микропроцессор позволяет *масштабировать индекс*. Это означает, что если указать после имени индексного регистра знак умножения «*» с последующей цифрой 2, 4 или 8, то содержимое индексного регистра будет умножаться на 2, 4 или 8, то есть масштабироваться. Применение масштабирования облегчает работу с массивами, которые имеют размер элементов, равный 2, 4 или 8 байтам, так как микропроцессор сам производит коррекцию индекса для получения адреса очеред-

ного элемента массива. Нам нужно лишь загрузить в индексный регистр значение требуемого индекса (считая от 0). Кстати сказать, возможность масштабирования появилась в микропроцессорах Intel, начиная с модели i486. По этой причине в рассматриваемом ниже примере программы стоит директива .486. Ее назначение, как и ранее использовавшейся директивы .386, в том, чтобы указать ассемблеру учитывать дополнительные возможности новых микропроцессоров при формировании машинных команд.

В качестве примера использования масштабирования рассмотрим листинг 12.2, в котором просматривается массив, состоящий из слов, и производится сравнение этих элементов с нулем. Выводится соответствующее сообщение.

Листинг 12.2. Просмотр массива слов с использованием масштабирования

```
;prg_12_2.asm
MASM
MODEL    small
STACK     256
.data
;тексты сообщений:
mes1    db      "не равен 0!$",0ah,0dh
mes2    db      "равен 0!$",0ah,0dh
mes3    db      0ah,0dh,'Элемент $'
masdw   2,7,0,0,1,9,3,6,0,8          ;исходный массив
.code
.486
main:
    mov    ax,@data
    mov    ds,ax
    xor    ax,ax
prepare:
    mov    cx,10
    mov    esi,0
compare:
    mov    dx,mas[esi*2]
    cmp    dx,0
    je     equal
not_equal:
    mov    ah,09h
    lea    dx,mes3
    int    21h
    mov    ah,02h
    mov    dx,si
    add    dl,30h
    int    21h
    mov    ah,09h
    lea    dx,mes1
    int    21h
    inc    esi
    dec    cx
    jcxz  exit
    jmp    compare
equal:
    ;нет – повторить цикл
    ;равно 0
;начало сегмента данных
;это обязательно
;значение счетчика цикла в cx
;индекс в esi
;первый элемент массива в dx
;сравнение dx с 0
;переход, если равно
;не равно
;вывод сообщения на экран
;вывод номера элемента массива на экран
;на следующий элемент
;условие для выхода из цикла
;cx=0? Если да – на выход
```

```

mov ah,09h ;вывод сообщения mes3 на экран
lea dx,mes3
int 21h
mov ah,02h
mov dx,si
add dl,30h
int 21h
mov ah,09h      ;вывод сообщения mes2 на экран
lea dx,mes2
int 21h
inc esi        ;на следующий элемент
dec cx          ;все элементы обработаны?
jcxz exit
jmp compare

exit:
    mov ax,4c00h    ;стандартный выход
    int 21h
end main         ;конец программы

```

Еще несколько слов о соглашениях:

- Если для описания адреса используется только один регистр, то речь идет о базовой адресации, и этот регистр рассматривается как базовый:
 $\text{:переслать байт из области данных,}$
 $\text{:адрес которой находится в регистре ebx:}$
 mov al,[ebx]
- Если для задания адреса в команде используется прямая адресация (в виде идентификатора) в сочетании с одним регистром, то речь идет об индексной адресации. Регистр считается индексным, и поэтому можно использовать масштабирование для получения адреса нужного элемента массива:
 $\text{add eax,mas[ebx*4]}$
 $\text{:сложить содержимое eax с двойным словом}$
 $\text{:в памяти по адресу mas + (ebx)*4}$
- Если для описания адреса используются два регистра, то речь идет о базово-индексной адресации. Левый регистр рассматривается как базовый, а правый — как индексный. В общем случае это не принципиально, но если мы используем масштабирование с одним из регистров, то он всегда является индексным. Но лучше придерживаться определенных соглашений. Помните, что применение регистров ebp/bp и esp/sp по умолчанию подразумевает, что сегментная составляющая адреса находится в регистре ss.

Заметим, что базово-индексную адресацию не возбраняется сочетать с прямой адресацией или указанием непосредственного значения. Адрес тогда будет формироваться как сумма всех компонентов.

К примеру:

```

    mov ax,mas[ebx][ecx*2]
;адрес операнда равен [mas+(ebx)+(ecx)*2]

...
    sub dx,[ebx+8][ecx*4]
;адрес операнда равен [(ebx)+8+(ecx)*4]

```

Но имейте в виду, что масштабирование эффективно лишь тогда, когда размерность элементов массива равна 2, 4 или 8 байтам. Если же размерность элементов другая, то организовывать обращение к элементам массива нужно обычным способом, как описано выше.

Рассмотрим пример работы с массивом из пяти трехбайтовых элементов (листинг 12.3). Младший байт в каждом из этих элементов представляет собой некий счетчик, а старшие два байта — что-то еще, для нас не имеющее никакого значения. Необходимо последовательно обработать элементы данного массива, увеличив значения счетчиков на единицу.

Листинг 12.3. Обработка массива элементов с нечетной длиной

```
:prg_12_3.asm
MASM
MODEL small ;модель памяти
STACK 256 ;размер стека
.data ;начало сегмента данных
N=5 ;количество элементов массива
mas db 5 dup (3 dup (0)) ;сегмент кода
.code ;точка входа в программу
main:
    mov ax,@data
    mov ds,ax
    xor ax,ax ;обнуление ax
    mov si,0 ;0 в si
    mov cx,N ;N в cx
go:
    mov dl,mas[si] ;первый байт поля в dl
    inc dl ;увеличение dl на 1 (по условию)
    mov mas[si],dl ;заслать обратно в массив
    add si,3 ;сдвиг на следующий элемент массива
    loop go ;повтор цикла
    mov si,0 ;подготовка к выводу на экран
    mov cx,N
show: ;вывод на экран содержимого
       ;первых байт полей
    mov dl,mas[si]
    add dl,30h
    mov ah,02h
    int 21h
    loop show
exit:
    mov ax,4c00h ;стандартный выход
    int 21h
end main ;конец программы
```

Двумерные массивы

С представлением одномерных массивов в программе на ассемблере и организацией их обработки все достаточно просто. А как быть, если программа должна обрабатывать двумерный массив? Все проблемы возникают по-прежнему из-за

того, что специальных средств для описания такого типа данных в ассемблере нет. Двумерный массив нужно моделировать. На описании самих данных это почти никак не отражается — память под массив выделяется с помощью директивы `reserved` и инициализации памяти. Непосредственно моделирование обработки массива производится в сегменте кода, где программист, описывая алгоритм обработки ассемблеру, определяет, что некоторую область памяти необходимо трактовать как двумерный массив. При этом вы вольны в выборе того, как понимать расположение элементов двумерного массива в памяти: по строкам или по столбцам. Если последовательность однотипных элементов в памяти трактуется как двумерный массив, расположенный по строкам, то адрес элемента (i, j) вычисляется по формуле:

`(база + количество_элементов_в_строке * размер_элемента * i+j)`

Здесь $i = 0 \dots n-1$ указывает номер строки, а $j = 0 \dots m-1$ указывает номер столбца. Например, пусть имеется массив чисел (размером в 1 байт) `mas(i, j)` с размерностью 4×4 ($i = 0 \dots 3$, $j = 0 \dots 3$):

| | | | |
|----|----|----|----|
| 23 | 04 | 05 | 67 |
| 05 | 06 | 07 | 99 |
| 67 | 08 | 09 | 23 |
| 87 | 09 | 00 | 08 |

В памяти элементы этого массива будут расположены в следующей последовательности:

23 04 05 67 05 06 07 99 67 08 09 23 87 09 00 08

Если мы хотим трактовать эту последовательность как двумерный массив, приведенный выше, и извлечь, например, элемент `mas(2, 3) = 23`, то, проведя нехитрый подсчет, убедимся в правильности наших рассуждений:

Эффективный адрес `mas(2, 3) = mas + 4 * 1 * 2 + 3 = mas + 11`

Посмотрите на представление массива в памяти и убедитесь, что по этому смещению действительно находится нужный элемент массива.

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

○ сочетание прямого адреса, как базового компонента адреса, и двух индексных регистров для хранения индексов:

`mov ax,[mas+esi][ebx]`

○ сочетание двух индексных регистров, один из которых является и базовым, и индексным одновременно, а другой — только индексным:

`mov ax,[ebx+esi][eax]`

В программе это будет выглядеть примерно так:

```
;Фрагмент программы выборки элемента
;массива mas(2,3) и его обнуления
.data
masdb 23,4,5,67,5,6,7,99,67,8,9,23,87,9,0,8
```

```
i=2  
j=3  
.code
```

```
...  
    mov si,4*i*i  
    mov di,j  
    mov al,mas[si][di]      ;в а1 элемент mas(2,3)  
...
```

В качестве законченного примера рассмотрим программу поиска элемента в двумерном массиве чисел (листинг 12.4). Элементы массива заданы статически.

Листинг 12.4. Поиск элемента в двумерном массиве

```
;prg_12_4.asm  
MASM  
MODEL small  
STACK 256  
.data  
;матрица размером 2x5 – если ее не инициализировать,  
;то для наглядности она может быть описана так:  
;array dw 2 DUP (5 DUP (??))  
;но мы ее инициализируем:  
array dw 1,2,3,4,5,6,7,3,9,0  
;логически это будет выглядеть так:  
;array= {1 2}  
; {3 4}  
; {5 6}  
; {7 3}  
; {9 0}  
    elem dw 3           ;элемент для поиска  
faileddb 0ah,0dh,'Нет такого элемента в массиве!','$'  
success db 0ah,0dh,'Такой элемент в массиве присутствует ","'$'  
foundtime db ?          ;количество найденных элементов  
fnd db " раз(a)",0ah,0dh,'$'  
.code  
main:  
    mov ax,@data  
    mov ds,ax  
    xor ax,ax  
    mov si,0           ;si=столбцы в матрице  
    mov bx,0           ;bx=строки в матрице  
    mov cx,5           ;число для внешнего цикла (по строкам)  
external:  
    mov ax,array[bx][si] ;в ах первый элемент матрицы  
    push cx            ;сохранение в стеке счетчика внешнего цикла  
    mov cx,2            ;число для внутреннего цикла (по столбцам)  
internal:             ;внутренний цикл по строкам  
    inc si              ;передвижение на следующий элемент в строке  
;сравниваем содержимое текущего элемента в ах с искомым элементом:  
    cmp ax,elem  
;если текущий совпал с искомым, то переход на here для обработки,  
;иначе – цикл продолжения поиска
```

```

je      here
;иначе – цикл по строке cx=2 раз
loop    internal

here:
    jcxz  move_next      ;просмотрели строку?
    inc   foundtime       ;иначе – увеличиваем счетчик совпадших
move_next:
    pop   cx              ;продвижение в матрице
    add   bx,1             ;восстанавливаем CX из стека (5)
    loop  external         ;передвигаемся на следующую строку
    cmp   foundtime,0h    ;цикл (внешний)
    ja    eql              ;сравнение числа совпадших с 0
    not_equal:
        mov   ah,09h          ;если больше 0, то переход
        mov   dx,offset failed ;нет элементов, совпадших с искомым
        int  21h              ;вывод сообщения на экран
        jmp  exit              ;на выход
eq1:
        mov   ah,09h          ;есть элементы, совпадшие с искомым
        mov   dx,offset success ;вывод сообщений на экран
        int  21h
        mov   ah,02h
        mov   dl,foundtime
        add   dl,30h
        int  21h
        mov   ah,09h
        mov   dx,offset fnd
        int  21h
exit:
        mov   ax,4c00h          ;выход
        int  21h              ;стандартное завершение программы
end   main                  ;конец программы

```

При анализе работы программы не забывайте, что в языке ассемблера принято элементы массива нумеровать с 0. При поиске определенного элемента массив просматривается от начала и до конца. Программа сохраняет в поле `foundtime` количество вхождений искомого элемента в массив. В качестве индексных регистров используются `si` и `bx`.

Типовые операции с массивами

Для демонстрации основных приемов работы с массивами лучше всего подходят программы поиска или сортировки. Рассмотрим одну такую программу, выполняющую сортировку массива по возрастанию (листинг 12.5).

Листинг 12.5. Сортировка массива

```

<1>      :prg_12_5.asm
<2>      MASM
<3>      MODEL small
<4>      STACK 256
<5>      .data

```

Листинг 12.5 (продолжение)

```
<6>    mes1 db 0ah,0dh,'Исходный массив - $',0ah,0dh
<7>    ;некоторые сообщения
<8>    mes2 db 0ah,0dh,'Отсортированный массив - $',0ah,0dh
<9>    n equ 9           ;количество элементов в массиве, считая с 0
<10>   mas dw 2,7,4,0,1,9,3,6,5,8      ;исходный массив
<11>   tmp dw 0          ;переменные для работы с массивом
<12>   i dw 0
<13>   j dw 0
<14> .code
<15> main:
<16>     mov ax,@data
<17>     mov ds,ax
<18>     xor ax,ax
<19>     ;вывод на экран исходного массива
<20>     mov ah,09h
<21>     lea dx,mes1
<22>     int 21h           ;вывод сообщения mes1
<23>     mov cx,10
<24>     mov si,0
<25> show_primary:        ;вывод значения элементов
<26>                      ;исходного массива на экран
<27>     mov dx,mas[si]
<28>     add dl,30h
<29>     mov ah,02h
<30>     int 21h
<31>     add si,2
<32>     loop show_primary
<33>
<34> ;строки 40–85 программы эквивалентны следующему коду на языке С:
<35> ;for (i=0;i<9;i++)
<36>   ;  for (j=9;j>i;j--)
<37>   ;    if (mas[i]>mas[j])
<38>   ;    {tmp=mas[i];
<39>   ;     mas[i]=mas[j];
<40>   ;     mas[j]=tmp;}
<41>   mov i,0            ;инициализация i
<42>                   ;внутренний цикл по j
<43> internal:
<44>     mov j,9            ;инициализация j
<45>     jmp cycl_j         ;переход на тело цикла
<46> exchange:
<47>     mov bx,i  ;bx=i
<48>     shl bx,1
<49>     mov ax,mas[bx]  ;ax=mas[i]
<50>     mov bx,j  ;bx=j
<51>     shl bx,1
<52>     cmp ax,mas[bx]  ;mas[i] ? mas[j] – сравнение элементов
<53>     jle lesser        ;если mas[i] меньше, то обмен не нужен
<54>                   ;и переход на продвижение далее по массиву
<55>                   ;иначе tmp=mas[i], mas[i]=mas[j], mas[j]=tmp;
<56>                   ;tmp=mas[i]
```

```

<56>          mov  bx,i      :bx=i
<57>          shl  bx,1      :умножаем на 2, так как элементы - слова
<58>          mov  tmp,ax    :tmp=mas[i]

<59>
<60>          mov  bx,j      :mas[i]=mas[j]
<61>          shl  bx,1      :bx=j
<62>          mov  ax,mas[bx]  :умножаем на 2, так как элементы - слова
<63>          mov  bx,i      :ax=mas[j]
<64>          shl  bx,1      :bx=i
<65>          mov  mas[bx],ax  :умножаем на 2, так как элементы - слова
<66>          mov  mas[i],mas[j] :mas[i]=mas[j]

<67>
<68>          mov  bx,j      :mas[j]=tmp
<69>          shl  bx,1      :bx=j
<70>          mov  ax,tmp    :умножаем на 2, так как элементы - слова
<71>          mov  ax,tmp    :ax=tmp
<72>          mov  mas[bx],ax  :mas[j]=tmp
<73> lesser:
<74>          dec  j       :j-
<75>          cycl_j:
<76>          mov  ax,j      :тело цикла по j
<77>          cmp  ax,i      :ax=j
<78>          jg   exchange  :сравнить j ? i
<79>          jg   exchange  :если j>i, то переход на обмен
<80>          jg   exchange  :иначе на внешний цикл по i
<81>          inc  i       :i++
<82>          cmp  i,n      :сравнить i ? n - прошли до конца массива
<83>          jl   internal :если i<n - продолжение обработки
<84>
<85> ;вывод отсортированного массива
<86>          mov  ah,09h
<87>          lea  dx,mes2
<88>          int  21h
<89> prepare:
<90>          mov  cx,10
<91>          mov  si,0
<92> show:           ;вывод значения элемента на экран
<93>          mov  dx,mas[si]
<94>          add  dl,30h
<95>          mov  ah,02h
<96>          int  21h
<97>          add  si,2
<98>          loop show
<99> exit:
<100>         mov  ax,4c00h  :стандартный выход
<101>         int  21h
<102> end  main      :конец программы

```

В основе программы лежит алгоритм, похожий на метод пузырьковой сортировки. Эта программа не претендует на безусловную оптимальность, так как существует целая теория, касающаяся подобного типа сортировок. Перед нами стоит

другая цель — показать использование средств ассемблера для решения подобного рода задач. В программе два цикла. Внешний цикл определяет позицию в массиве очередного элемента, с которым производится попарное сравнение элементов правой части массива (относительно этого элемента). За каждую итерацию внешнего цикла на месте этого очередного элемента оказывается меньший элемент из правой части массива (если он есть). В остальном программа достаточно проста и на языке высокого уровня заняла бы около десятка строк.

Структуры

Рассмотренные нами выше массивы представляют собой совокупность однотипных элементов. Но часто в приложениях возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип. Это очень актуально, например, для программ баз данных, где необходимо связывать совокупность данных разного типа с одним объектом. К примеру, выше мы рассмотрели листинг 12.3, в котором работа производилась с массивом трехбайтовых элементов. Каждый элемент, в свою очередь, представлял собой два элемента разных типов: однобайтовое поле счетчика и двухбайтовое поле, которое могло нести еще какую-то нужную для хранения и обработки информацию. Если читатель знаком с одним из языков высокого уровня, то он знает, что такой объект обычно описывается с помощью специального типа данных — структуры. С целью повысить удобство использования языка ассемблера в него также был введен такой тип данных.

По определению *структура* — это тип данных, состоящий из фиксированного числа элементов разного типа.

Для использования структур в программе необходимо выполнить три действия:

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

Очень важно, чтобы вы с самого начала уяснили, в чем разница между *описанием* структуры в программе и ее *определением*. Описать структуру в программе означает лишь указать ее схему или шаблон; память при этом не выделяется. Этот шаблон можно рассматривать лишь как информацию для транслятора о расположении полей и их значении по умолчанию. Определить структуру — значит дать указание транслятору выделить память и присвоить этой области памяти символическое имя. Описать структуру в программе можно только один раз, а определить — любое количество раз.

Описание шаблона структуры

Описание шаблона структуры имеет следующий синтаксис:

```
имя_структуре STRUC  
<описание полей>  
имя_структуре ENDS
```

Здесь *<описание полей>* представляет собой последовательность директив описания данных db, dw, dd, dq и dt. Их операнды определяют размер полей и при необходимости начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры.

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

Рассмотрим работу со структурами на примере моделирования базы данных о сотрудниках некоторого отдела. Для простоты, чтобы уйти от проблем преобразования информации при вводе, условимся, что все поля символьные. Определим структуру записи этой базы данных следующим шаблоном:

```
worker struc ;информация о сотруднике  
nam db 30 dup (" ") ;фамилия, имя, отчество  
sex db " " ;пол  
position db 30 dup (" ") ;должность  
age db 2 dup (" ") ;возраст  
standing db 2 dup (" ") ;стаж  
salary db 4 dup (" ") ;оклад в рублях  
birthdate db 8 dup (" ") ;дата рождения  
worker ends
```

Определение данных с типом структуры

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

[имя переменной] имя_структуре <[список значений]>

Здесь:

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

список значений — заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми. Его задание также

необязательно. Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если такие заданы. Допускается инициализация отдельных полей, но в этом случае пропущенные поля должны отделяться запятыми. Пропущенные поля будут инициализированы значениями из шаблона структуры. Если при определении новой переменной с типом данной структуры мы согласны со всеми значениями полей в ее шаблоне (то есть заданными по умолчанию), то нужно просто написать угловые скобки. К примеру:

victor worker <>.

Для примера определим несколько переменных с типом описанной выше структуры:

```
data      segment
sotr1    worker<"Гурко Андрей Вячеславович",,'художник',
'33','15','1800','26.01.64'
sotr2    worker<"Михайлова Наталья Геннадьевна",'ж','программист',
'30','10','1680','27.10.58'
sotr3    worker<"Степанов Юрий Лонгинович",,'художник',
'38','20','1750','01.01.58'
sotr4    worker<"Юрова Елена Александровна",'ж','связист', '32','2','09.01.66'
sotr5    worker<> ;здесь все значения по умолчанию
data      ends
```

Методы работы со структурой

Идея введения структурного типа в любой язык программирования состоит в объединении разнотипных переменных в один объект. В языке должны быть средства доступа к этим переменным внутри конкретного экземпляра структуры. Для того чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор — символ «.» (точка). Он используется в следующей синтаксической конструкции:

адресное_выражение.имя_поля_структурь

Здесь:

адресное_выражение — идентификатор некоторого структурного типа или выражение в скобках в соответствии с указанными ниже синтаксическими правилами (рис. 12.1);

имя_поля_структурь — имя поля из шаблона структуры. Это на самом деле тоже адрес, а точнее, смещение поля от начала структуры.

Таким образом оператор . вычисляет выражение:

(адресное_выражение) + (имя_поля_структурь)

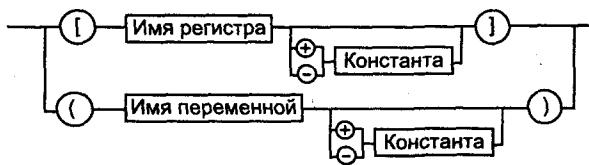


Рис. 12.1. Синтаксис адресного выражения в операторе обращения к полю структуры

Продемонстрируем на примере определенной нами структуры `worker` некоторые приемы работы со структурами. К примеру, требуется извлечь в `ax` значения поля с возрастом. Так как вряд ли возраст трудоспособного человека будет больше величины 99 лет, то после помещения содержимого этого символьного поля в регистр `ax` его будет удобно преобразовать в двоичное представление командой `aad` (см. урок 8). Будьте внимательны, так как из-за принципа хранения данных «младший байт по младшему адресу» старшая цифра возраста будет помещена в `al`, а младшая — в `ah`. Для корректировки достаточно использовать команду `xchg al, ah`:

```
    mov  ax,word ptr sotr1.age      ;в al возраст sotr1
    xchg ah,al
;а можно и так:
    lea   bx,sotr1
    mov  ax,word ptr [bx].age
    xchg ah,al
```

Давайте представим, что сотрудников не четверо, а намного больше, и к тому же их число и информация о них постоянно меняются. В этом случае теряется смысл явного определения переменных с типом `worker` для конкретных личностей. Язык ассемблера разрешает определять не только отдельную переменную с типом структуры, но и массив структур. К примеру, определим массив из 10 структур типа `worker`:

```
mas_sotr worker 10 dup (<>)
```

Дальнейшая работа с массивом структур производится так же, как и с одномерным массивом. Здесь возникает несколько вопросов. Как быть с размером и как организовать индексацию элементов массива?

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

```
worker  struc
...
worker  ends
...
mas_sotr worker 10 dup (<>)
...
    mov  bx,type    worker ;bx=77
    lea   di,mas_sotr
;извлечь и вывести на экран пол всех сотрудников:
    mov  cx,10
cycl:
    mov  dl,[di].sex
...          ;вывод на экран содержимого
;поля sex структуры worker
    add  di,bx      ;к следующей структуре в массиве mas_sotr
loop cycl
```

Как выполнить копирование поля из одной структуры в соответствующее поле другой структуры? Или как выполнить копирование всей структуры? Давайте выполним копирование поля `nam` третьего сотрудника в поле `nam` пятого сотрудника:

```
worker struc  
...  
worker ends  
...  
mas_sotr worker 10 dup (<>)  
...  
    mov bx,offset mas_sotr  
    mov si,(type worker)*2 ;si=77*2  
    add si,bx  
    mov di,(type worker)*4 ;si=77*4  
    add di,bx  
    mov cx,30  
rep movsb
```

На прилагаемой к книге дискете в каталоге ..\lesson12\struct\ приведена программа, которая осуществляет работу с базой данных о сотрудниках. На ее примере вы можете глубже познакомиться с тем, как организовать работу со структурами в своей программе. Возможно, для читателя имеет смысл в полном объеме исследовать работу этой программы после знакомства с макрокомандами на следующем уроке.

Объединения

Мне кажется, что ремесло программиста рано или поздно делает человека похожим на хорошую домохозяйку. Он, подобно ей, постоянно находится в поиске: где бы чего-нибудь сэкономить, урезать, из минимума продуктов сделать прекрасный обед. И если это удается, то и моральное удовлетворение получается ничуть не меньше, а может, и больше, чем от прекрасного обеда у домохозяйки. Степень этого удовлетворения, как мне кажется, зависит от степени любви к своей профессии. С другой стороны, успехи в разработке программного и аппаратного обеспечения несколько расслабляют программиста, и довольно часто наблюдается ситуация, похожая на известную пословицу про мууху и слона, — для решения некоторой мелкой задачи привлекаются тяжеловесные средства, эффективность которых в общем случае значима только при реализации сравнительно больших проектов.

Наличие в языке следующих двух типов данных, наверное, объясняется стремлением хозяйки максимально эффективно использовать рабочую площадь стола (оперативной памяти) при приготовлении еды или для размещения продуктов (данных программы).

Представим ситуацию, когда мы используем некоторую область памяти для размещения некоторого объекта программы (переменной, массива или структуры). Вдруг после некоторого этапа работы у нас отпадет надобность в использовании этих данных. Обычно память остается занятой до конца работы программы.

Конечно, в принципе, ее можно было бы использовать для хранения других переменных, но при этом без принятия специальных мер нельзя изменить тип и имя. Неплохо было бы иметь возможность переопределить эту область памяти для объекта с другим типом и именем. Язык ассемблера предоставляет такую возможность в виде специального типа данных, называемого объединением. *Объединение* — тип данных, позволяющий трактовать одну и ту же область памяти как имеющую разные типы и имена.

Описание объединений в программе напоминает описание структур, то есть сначала описывается шаблон, в котором с помощью директив описания данных перечисляются имена и типы полей:

имя_объединения UNION

<описание полей>

имя_объединения ENDS

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

Листинг 12.6, который мы сейчас рассмотрим, примечателен тем, что, кроме демонстрации использования собственно типа данных «объединение», в нем показывается возможность взаимного вложения структур и объединений. Постарайтесь внимательно отнести к анализу этой программы. Основная идея здесь в том, что указатель на память, формируемый программой, может быть представлен в виде:

- 16-битного смещения;
- 32-битного смещения;
- пары из 16-битного смещения и 16-битной сегментной составляющей адреса;
- пары из 32-битного смещения и 16-битного селектора.

Какие из этих указателей можно применять в конкретной ситуации, зависит от режима адресации (use16 или use32) и режима работы микропроцессора. Так вот, описанный в листинге 12.6 шаблон объединения позволяет нам облегчить формирование и использование указателей различных типов.

Листинг 12.6. Пример использования объединения

```
masm
model small
stack 256
.586P
pnt  struc ;структура pnt, содержащая вложенное объединение
      union    ;описание вложенного в структуру объединения
      offs_16   dw    ?
      offs_32   dd    ?
ends    ;конец описания объединения
segm   dw    ?
ends    ;конец описания структуры
```

```

.data
point union ;определение объединения,
;содержащего вложенную структуру
off_16 dw ?
off_32 dd ?
point_16 prt <>
point_32 prt <>
point ends
tst db «Строка для тестирования»
adr_data point <> ;определение экземпляра объединения
.code

main:
    mov ax,@data
    mov ds,ax
    mov ax,seg tst
;записать адрес сегмента строки tst в поле структуры adr_data
    mov adr_data.point_16.segm,ax
;когда понадобится, можно извлечь значение из этого поля обратно,
;к примеру, в регистр bx:
    mov bx,adr_data.point_16.segm
;формируем смещение в поле структуры adr_data
    mov ax,offset tst ;смещение строки в ах
    mov adr_data.point_16.offset,ax
;аналогично, когда понадобится, можно извлечь
;значение из этого поля:
    mov bx,adr_data.point_16.offset
exit:
    mov ax,4c00h
    int 21h
end main

```

Когда вы будете работать в защищенном режиме микропроцессора и использовать 32-разрядные адреса, то аналогичным способом можете заполнить и использовать описанное выше объединение.

Записи

Наша «хозяйка-программист» становится все более экономной. Она уже хочет работать с продуктами на молекулярном уровне, без любых отходов и напрасных трат. Подумаем, зачем тратить под некоторый программный индикатор со значением «включено-выключено» целых восемь разрядов, если вполне хватает одного? А если таких индикаторов несколько, то расход оперативной памяти может стать весьма ощутимым.

Когда мы знакомились с логическими командами, то говорили, что их можно применять для решения подобной проблемы. Но это не совсем эффективно, так как велика вероятность ошибок, особенно при составлении битовых масок. TASM предоставляет нам специальный тип данных, использование которого помогает решить проблему работы с битами более эффективно. Речь идет о специальном

типе данных — записях. Запись — структурный тип данных, состоящий из фиксированного числа элементов длиной от одного до нескольких бит.

При описании записи для каждого элемента указывается его длина в битах и, что необязательно, некоторое значение. Суммарный размер записи определяется суммой размеров ее полей и не может быть более 8, 16 или 32 бит. Если суммарный размер записи меньше указанных значений, то все поля записи «прижимаются» к младшим разрядам.

Использование записей в программе, так же, как и структур, организуется в три этапа:

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

Описание записи

Описание шаблона записи имеет следующий синтаксис:

имя_записи RECORD <описание элементов>

Здесь:

<описание элементов> представляет собой последовательность описаний отдельных элементов записи согласно синтаксической диаграмме (рис. 12.2).

При описании шаблона память не выделяется, так как это всего лишь информация для транслятора ассемблера о структуре записи. Так же как и для структур, местоположение шаблона в программе может быть любым, но при этом необходимо учитывать логику работы однопроходного транслятора.

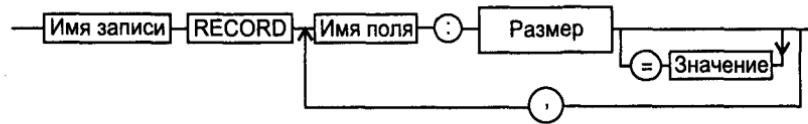


Рис. 12.2. Синтаксис описания шаблона записи

Определение экземпляра записи

Для использования шаблона записи в программе необходимо определить переменную с типом данной записи, для чего применяется следующая синтаксическая конструкция (рис. 12.3).

Анализируя эту синтаксическую диаграмму, можно сделать вывод, что инициализация элементов записи осуществляется достаточно гибко. Рассмотрим несколько вариантов инициализации.

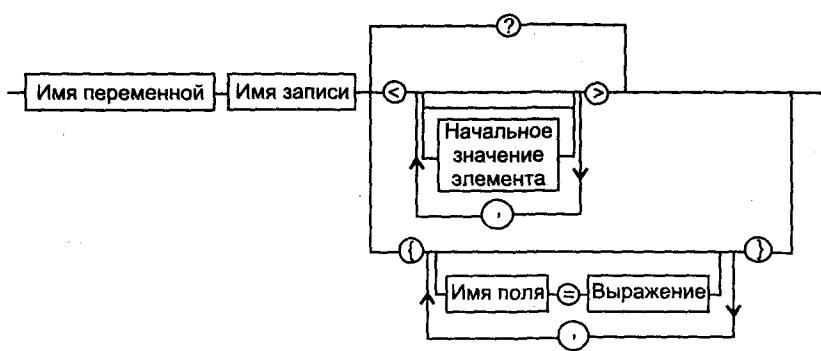


Рис. 12.3. Синтаксис описания экземпляра записи

Если инициализировать поля не требуется, то достаточно указать ? при определении экземпляра записи:

```
...
iotest record i1:1, i2:2=11, i3:1, i4:2=11, i5:2=00
...
flag iotest?
```

Если вы составите и исследуете в отладчике тестовый пример с данным определением записи, то увидите, что все поля переменной типа запись flag обнуляются. Это происходит несмотря на то, что в определении записи заданы начальные значения полей.

Если требуется частичная инициализация элементов, то они заключаются в угловые (< и >) или фигурные ({ и }) скобки. Различие здесь в том, что в угловых скобках элементы должны быть заданы в том же порядке, что и в определении записи. Если значение некоторого элемента совпадает с начальным, то его можно не указывать, но обязательно обозначить его запятой. Для последних элементов идущие подряд запятые можно опустить.

К примеру, согласиться со значениями по умолчанию можно так:

```
iotest record i1:1, i2:2=11, i3:1, i4:2=11, i5:2=00
...
flag iotest<> ;согласились со значением по умолчанию
```

Изменить значение поля i2 можно так:

```
iotest record i1:1, i2:2=11, i3:1, i4:2=11, i5:2=00
```

```
...
flag iotest<,10,>; переопределили i2
```

Применяя фигурные скобки, также можно указать выборочную инициализацию полей, но при этом необязательно обозначать запятыми поля, со значениями по умолчанию которых мы согласны:

```
iotest record i1:1, i2:2=11, i3:1, i4:2=11, i5:2=00
```

```
...
flag iotest{ i2=10 } ;переопределили i2, не обращая
;внимания на порядок
;следования других компонентов записи
```

Работа с записями

Как организовать работу с отдельными элементами записи? Обычные механизмы адресации здесь бессильны, так как они работают на уровне ячеек памяти, то есть байтов, а не отдельных битов. Здесь программисту нужно приложить некоторые усилия. Прежде всего для понимания проблемы нужно усвоить несколько моментов:

- Каждому имени элемента записи ассемблер присваивает числовое значение, равное количеству сдвигов вправо, которые нужно произвести для того, чтобы этот элемент оказался «прижатым» к началу ячейки. Это дает нам возможность локализовать его и работать с ним. Но для этого нужно знать длину элемента в битах.
- Сдвиг вправо производится с помощью команды сдвига `shr`.
- Ассемблер содержит оператор `width`, который позволяет узнать размер элемента записи в битах или полностью размер записи. Варианты применения оператора `width`:
 - `width имя_элемента_записи` — значением оператора будет размер элемента в битах;
 - `width имя_экземпляра_записи` или `width имя_типа_записи` — значением оператора будет размер всей записи в битах.

```
mov al, width i2
...
mov ax, width iotest
```
- Ассемблер содержит оператор `mask`, который позволяет локализовать биты нужного элемента записи. Эта локализация производится путем создания маски, размер которой совпадает с размером записи. В этой маске обнулены биты во всех позициях, за исключением тех, которые занимает элемент в записи.
- Сами действия по преобразованию элементов записи производятся с помощью логических команд.

Теперь у вас есть вся информация о средствах ассемблера для работы с записями. Вы также поняли, что непосредственно обратиться к элементу записи невозможно. Чтобы произвести обработку интересующего нас элемента, нужно сначала выделить его, сдвинуть при необходимости к младшим разрядам, выполнить требуемые действия и поместить обратно на свое место в записи. Поэтому, чтобы вам не изобретать каждый раз велосипед, далее мы опишем типовые алгоритмы осуществления этих операций над элементами записи. Ваша задача — закодировать эти алгоритмы тем или иным способом в соответствии с требованиями задачи.

Для выделения элемента записи:

- Поместить запись во временную память — регистр (8-, 16- или 32-битный, в зависимости от размера записи).
- Получить битовую маску, соответствующую элементу записи, с помощью оператора `mask`.

- Локализовать биты в регистре с помощью маски и команды `and`.
 - Сдвинуть биты элемента к младшим разрядам регистра командой `shr`. Число разрядов для сдвига получить с использованием имени элемента записи.
- В результате этих действий элемент записи будет локализован в начале рабочего регистра и далее с ним можно производить любые действия (как, см. ниже).
- Как мы уже выяснили, с элементами записи производятся любые действия, как над обычной двоичной информацией. Единственное, что нужно отслеживать, — это размер битового поля. Если, к примеру, размер поля увеличится, то впоследствии может произойти случайное изменение соседних полей битов. Поэтому желательно исключить изменение размера поля.

Чтобы поместить измененный элемент на его место в запись:

- Используя имя элемента записи в качестве счетчика сдвигов, сдвинуть влево биты элемента записи.
- Если вы не уверены в том, что разрядность результата преобразований не превысила исходную, можно выполнить «обрезание» лишних битов, используя команду `and` и маску элемента.
- Подготовить исходную запись к вставке измененного элемента путем обнуления битов в записи на месте этого элемента. Это можно сделать путем наложения командой `and` инвертированной маски элемента записи на исходную запись.
- С помощью команды `or` наложить значение в регистре на исходную запись.

В качестве примера рассмотрим листинг 12.7, который обнуляет поле `i2` в записи `iotest`.

Листинг 12.7. Работа с полем записи

```
:prg_12_7.asm
masm
model    small
stack    256
iotest   record i1:1, i2:2=11, i3:1, i4:2=11, i5:2=00
.data
flag     iotest<>
.code
main:
    mov ax,@data
    mov ds,ax
    mov al,mask i2
    shr al,i2      ;биты i2 в начале ах
    and al,0fch    ;обнулили i2
;помещаем i2 на место
    shl al,i2
    mov bl,[flag]
    xor bl,mask i2 ;сбросили i2
    or bl,al       ;наложили
    exit:
    mov ax,4c00h    ;стандартный выход
    int 21h
end main           ;конец программы
```

В заключение еще раз проанализируйте тип записи и особенности работы с ним. При этом обратите внимание на то обстоятельство, что мы нигде явно не просчитываем расположение битов. Поэтому, если понадобится изменить размер элемента или его начальное значение, достаточно внести изменения в экземпляр записи или в описание ее типа; функциональную часть программы, работающую с этой записью, трогать не нужно.

Дополнительные возможности обработки

Понимая важность для эффективного программирования такого типа данных, как запись, разработчики транслятора TASM, начиная с версии 3.0, включили в систему его команд две дополнительные команды на правах директив. Последнее означает, что эти команды внешне имеют формат обычных команд ассемблера, но после трансляции они приводятся к одной или нескольким машинным командам. Введение этих команд в язык TASM повышает наглядность работы с записями, оптимизирует код и уменьшает размер программы. Эти команды позволяют скрыть от программиста действия по выделению и установке отдельных полей записи (мы их обсуждали выше).

Для установки значения некоторого поля записи используется команда `setfield` с синтаксисом

`setfield имя_элемента_записи назначение, регистр_источник`

Для выборки значения некоторого поля записи используется команда `getfield` с синтаксисом

`getfield имя_элемента_записи регистр_назначение, источник`

Работа команды `setfield` заключается в следующем. Местоположение записи определяется операндом `назначение`, который может представлять собой имя регистра или адрес памяти. Операнд `имя_элемента_записи` определяет элемент записи, с которым ведется работа (по сути, если вы были внимательны, он определяет смещение элемента в записи относительно младшего разряда). Новое значение, в которое необходимо установить указанный элемент записи, должно содержаться в операнде `регистр_источник`. Обрабатывая данную команду, транслятор генерирует последовательность команд, которые выполняют следующие действия:

- сдвиг содержимого `регистр_источник` влево на количество разрядов, соответствующее расположению элемента в записи;
- логическую операцию `or` над операндами `назначение` и `регистр_источник`. Результат операции помещается в операнд `назначение`.

Важно отметить, что `setfield` не производит предварительной очистки элемента, в результате после логического сложения командой `or` возможно наложение старого содержимого элемента и нового устанавливаемого значения. Поэтому требуется предварительно подготовить поле в записи путем его обнуления.

Действие команды `getfield` обратно `setfield`. В качестве операнда `источник` может быть указан либо регистр, либо адрес памяти. В регистр, указанный операндом `регистр_назначение`, помещается результат работы команды — значение элемента записи. Интересная особенность связана с `регистр_назначение`. Команда `getfield`

всегда использует 16-битный регистр, даже если вы укажете в этой команде имя 8-битного регистра.

В качестве примера применения команд `setfield` и `getfield` рассмотрим листинг 12.8.

Листинг 12.8. Работа с полями записи

```
;prg_12_8.asm
masm
model    small
stack    256
iotest   record i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
.data
flag     iotest<>
.code
main:
    mov ax,@data
    mov ds,ax
    mov al,flag
    mov bl,3
    setfield i5 al,bl
    xor bl,bl
    getfield i5 bl,al
    mov bl,1
    setfield i4 al,bl
    setfield i5 al,bl
exit:
    mov ax,4c00h      ;стандартный выход
    int 21h
end main           ;конец программы
```

В листинге 12.8 демонстрируется порядок извлечения и установки некоторых полей записи. Результат работы команд `setfield` и `getfield` удобнее всего изучать в отладчике. При установке значений полей не производится их предварительная очистка. Это сделано специально. Для такого рода операций лучше использовать некоторые универсальные механизмы, иначе велик риск внесения ошибок, которые трудно обнаружить и исправить. В качестве такого механизма можно предложить макрокоманды, к рассмотрению которых мы и приступим на следующем уроке.

В заключение хотелось бы привести еще один пример использования записей. Это описание регистра `eflags`. Для удобства это описание мы разбили на три части: `eflags_1_7` — младший байт `eflags/flags`, `eflags_8_15` — второй байт `eflags/flags`, `eflags_h` — старшая половина `eflags`.

```
eflags_1_7  record sf7:1=0,zf6:1=0,c5:1=0,af4:1=0,c3:1=0,pf2:1=0,c1:=1,cf0:1=0
eflags_1_15 record c15:1=0,nt14:1=0,iopl:2=0,of11:1=0,df10:1=0,if9:1=1,tf8:1=0
eflags_h    record c:13=0,ac18:1=0,vm17:1=0,rf16:1=0
```

Запомните это описание. Когда вы освоите работу с макрокомандами и столкнетесь с необходимостью задействовать регистр флагов, то вы сразу же захотите написать соответствующую макрокоманду. Эта макрокоманда, если вы не забудете хорошо ее протестировать, избавит вас от многих трудно обнаруживаемых ошибок.

Подведем некоторые итоги:

- TASM поддерживает несколько дополнительных типов данных, значительно расширяющих возможности базовых директив резервирования и инициализации данных. По сути, эти типы заимствованы из языков высокого уровня и призваны облегчить разработку прикладных программ на ассемблере.
- Практическое использование дополнительных типов данных требует повышенной внимательности и отражает специфику программирования на языке ассемблера.
- Понятия массива и индексации массива весьма условны, и логическая интерпретация области памяти, отведенной под массив, определяется алгоритмом обработки.
- Тип структуры в языке ассемблера позволяет создать совокупность логически взаимосвязанных разнотипных данных и рассматривать их как отдельный объект. Это очень удобно в случаях, когда в программе необходимо иметь несколько таких объектов. В этом случае обычно организуют массив структур.
- Основное достоинство объединений — в возможности «плюрализма суждений» о типе одной и той же области памяти.
- Записи в языке ассемблера расширяют возможности логических команд для работы на уровне битов, что подчеркивает значение ассемблера как языка системного программирования.

13

УРОК

Макросредства языка ассемблера

-
- Понятие о макросредствах языка ассемблера
 - Псевдооператоры equ и =
 - Макрокоманды и макродирективы
 - Директивы условной компиляции
 - Директивы генерации ошибок пользователя
-

Любопытный читатель к данному уроку, вероятно, попытался самостоятельно написать хотя бы несколько программ на ассемблере. Скорее всего, эти программы были предназначены для решения небольших, чисто исследовательских задач, но даже на примере этих маленьких по объему программ вам, наверное, стали очевидны некоторые из нижеперечисленных проблем:

- плохое понимание исходного текста программы, особенно по прошествии некоторого времени после ее написания;
- ограниченность набора команд;
- повторяемость некоторых идентичных или незначительно отличающихся участков программы;
- необходимость включения в каждую программу участков кода, которые уже были использованы в других программах;
- и т. д.

Если бы мы писали программу на машинном языке, то данные проблемы были бы принципиально нерешаемыми. Но язык ассемблера, являясь символическим аналогом машинного языка, предоставляет для их решения ряд средств. Основной целью, которая при этом преследуется, является повышение удобства написания программ. В общем случае эта цель достигается по нескольким направлениям за счет:

- расширения набора директив;
- введения некоторых дополнительных команд, не имеющих аналогов в системе команд микропроцессора. За примером далеко ходить не нужно — команды `setfield` и `getfield`, рассмотренные на уроке 12. Они скрывают от программиста рутинные действия и генерируют наиболее эффективный код;
- введения сложных типов данных.

Но это все глобальные направления, по которым развивается сам транслятор от версии к версии. Что же делать программисту для решения его локальной задачи, для облегчения работы в определенной проблемной области? Для этого разработчики компиляторов ассемблера включают в язык и постоянно совершенствуют аппарат *макросредств*. Этот аппарат является очень мощным и важным. В общем случае есть смысл говорить о том, что транслятор ассемблера состоит из двух частей — непосредственно транслятора, формирующего объектный модуль, и *макроассемблера* (рис. 13.1). Если вы знакомы с языками С или С++, то, конечно,помните широко применяемый в них механизм препроцессорной обработки.

Он является некоторым аналогом механизма, заложенного в работу макроассемблера. Для тех, кто ничего раньше не слышал об этих механизмах, поясню их суть. Основная идея — использование подстановок, которые замещают определенным образом организованную символьную последовательность другой символьной последовательностью. Создаваемая таким образом последовательность может быть как последовательностью, описывающей данные, так и последовательностью программных кодов. Главное здесь то, что на входе макроассемблера может быть текст программы, весьма далекий по виду от программы на языке ассемблера, а на выходе обязательно будет текст на чистом ассемблере, содержащем символические аналоги команд системы машинных команд микропроцессора. Таким образом, обработка программы на ассемблере с использованием макросредств неявно осуществляется транслятором в две фазы (см. рис. 13.1). На первой фазе работает часть компилятора, называемая макроассемблером, основные функции которого мы описали выше. На второй фазе трансляции участвует непосредственно ассемблер, задачей которого является формирование объектного кода, содержащего текст исходной программы в машинном виде.

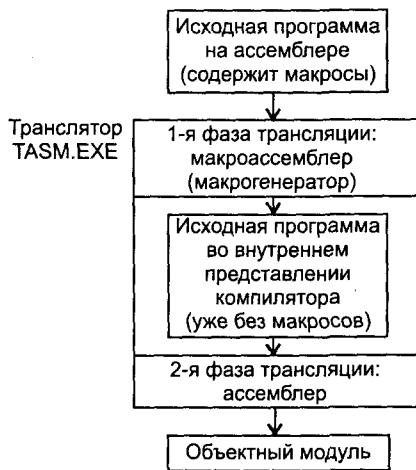


Рис. 13.1. Макроассемблер в общей схеме трансляции программы на TASM

Далее мы обсудим основной набор макросредств, доступных при использовании компилятора TASM. Отметим, что большинство этих средств доступно и в компиляторе с языка ассемблера фирмы Microsoft. Обсуждение начнем с простейших средств и закончим более сложными.

Псевдооператоры equ и =

К простейшим макросредствам языка ассемблера можно отнести псевдооператоры equ и «=» (равно). Их мы уже неоднократно использовали при написании программ. Эти псевдооператоры предназначены для присвоения некоторому выражению символьического имени или идентификатора. Впоследствии, когда в ходе трансляции этот идентификатор встретится в теле программы, макроассемблер

подставит вместо него соответствующее выражение. В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.

Синтаксис псевдооператора equ:

имя_иентификатора equ строка или числовое выражение

Синтаксис псевдооператора =:

имя_иентификатора = числовое выражение

Несмотря на внешнее и функциональное сходство, псевдооператоры equ и = отличаются следующим:

- из синтаксического описания видно, что с помощью equ иентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки, а псевдооператор = может использоваться только с числовыми выражениями;
- иентификаторы, определенные с помощью =, можно переопределять в исходном тексте программы, а определенные с использованием equ — нельзя.

Ассемблер всегда пытается вычислить значение строки, воспринимая ее как выражение. Для того чтобы строка воспринималась именно как текстовая, необходимо заключить ее в угловые скобки: <строка>. Кстати сказать, угловые скобки являются оператором ассемблера, с помощью которого транслятору сообщается, что заключенная в них строка должна трактоваться как текст, даже если в нее входят служебные слова ассемблера или операторы. Хотя в режиме Ideal это не обязательно, так как строка для equ в нем всегда трактуется как текстовая.

Псевдооператор equ удобно использовать для настройки программы на конкретные условия выполнения, замены сложных в обозначении объектов, многократно используемых в программе, более простыми именами и т. п. К примеру:

```
masm
model    small
stack     256
mas_size equ   10           ;размерность массива
akk equ    ax             ;переименовать регистр
mas_elem equ   mas[bx][si] ;адресовать элемент массива
.data
;описание массива из 10 байт:
masdb    mas_size dup (0)
.code
    mov   akk,@data          ;фактически mov ax,@data
    mov    ds,akk            ;фактически mov ds,ax
...
    mov    al,mas_elem        ;фактически mov al,mas[bx][si]
```

Псевдооператор = удобно использовать для определения простых абсолютных (то есть не зависящих от места загрузки программы в память) математических выражений. Главное условие то, чтобы транслятор мог вычислить эти выражения во время трансляции. К примеру:

```
.data
adr1    db    5 dup (0)
adr2    dw    0
```

```
len = 4
len = len+1 ;можно и так, через предыдущее определение
len = adr2-adr1
```

Как видно из примера, в правой части псевдооператора = можно использовать метки и ссылки на адреса — главное, чтобы в итоге получилось абсолютное выражение.

Компилятор TASM, начиная с версии 3.00, содержит директивы, значительно расширяющие его возможности по работе с текстовыми макросами. Эти директивы аналогичны некоторым функциям обработки строк в языках высокого уровня. Под *строками* здесь понимается текст, описанный с помощью псевдооператора equ. Набор этих директив следующий:

○ директива слияния строк catstr:

идентификатор catstr строка_1, строка_2, ... — значением этого макрояадреса будет новая строка, состоящая из сцепленной слева направо последовательности строк *строка_1, строка_2, ...*. В качестве сцепляемых строк могут быть указаны имена ранее определенных макрояадресов. К примеру:

```
preequ Привет,
name    equ   < Юля>
privet  catstr pre.name   ;privet= «Привет, Юля»
```

○ директива выделения подстроки в строке substr:

идентификатор substr строка, номер_позиции, размер — значением данного макрояадреса будет часть заданной строки, начинающаяся с позиции с номером *номер_позиции* и длиной, указанной в *размер*. Если требуется только остаток строки, начиная с некоторой позиции, то достаточно указать только номер позиции без указания размера. К примеру:

;продолжение предыдущего фрагмента:
privet catstr pre.name ;privet= «Привет, Юля»
name substr privet.7,3 ;name=<Юля»

○ директива определения вхождения одной строки в другую instr:

идентификатор instr номер_нач_позиции, строка_1, строка_2 — после обработки данного макрояадреса транслятором *идентификатору* будет присвоено числовое значение, соответствующее номеру (первой) позиции, с которой совпадают *строка_1* и *строка_2*. Если такого совпадения нет, то *идентификатор* получит значение 0;

○ директива определения длины строки в текстовом макрояадресе sizestr:

идентификатор sizestr строка — в результате обработки данного макрояадреса значение *идентификатор* устанавливается равным длине строки:

;как продолжение предыдущего фрагмента:
privet catstr pre.name ;privet= «Привет, Юля»
len sizestr privet ;len=10

Эти директивы очень удобно использовать при разработке *макрокоманд*, которые являются следующим макросредством, предоставляемым компилятором ассемблера.

Макрокоманды

Идеяно макрокоманда представляет собой дальнейшее развитие механизма замены текста. С помощью макрокоманд в текст программы можно вставлять последовательности строк (которые логически могут быть данными или командами) и даже более того — привязывать их к контексту места вставки.

Представим ситуацию, когда необходимо выполнить некоторые повторяющиеся действия. Программа из листинга 3.1 является ярким этому примером. Структурно в ней явно прослеживаются повторяющиеся участки кода. Их можно оформить в виде макрокоманд и использовать эти повторяющиеся фрагменты в различных программах. Дальнейшее наше обсуждение будет посвящено тому, как это сделать.

Определимся с терминологией. *Макрокоманда* представляет собой строку, содержащую некоторое символическое имя — *имя макрокоманды*, предназначенную для того, чтобы быть замещенной одной или несколькими другими строками. Имя макрокоманды может сопровождаться параметрами. Обычно программист сам чувствует момент, когда ему нужно использовать макрокоманды в своей программе. Если такая необходимость возникает и нет готового, ранее разработанного варианта нужной макрокоманды, то вначале необходимо задать ее *шаблон-описание*, который называют *макроопределением*. Синтаксис макроопределения следующий:

```
имя_макрокоманды тасго список_формальных_аргументов  
тело макроопределения  
endm
```

Где должны располагаться макроопределения? Есть три варианта:

- в начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы. Этот вариант следует применять в случаях, если определяемые вами макрокоманды актуальны только в пределах одной этой программы;
- в отдельном файле. Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву *include имя_файла*, к примеру:

```
masm  
model      small  
include    show.inc  
;в это место будет вставлен текст файла show.inc  
...
```

- в макробиблиотеке. Если у вас есть универсальные макрокоманды, которые используются практически во всех ваших программах, то их целесообразно записать в так называемую *макробиблиотеку*. Сделать актуальными макрокоманды из этой библиотеки можно с помощью все той же директивы *include*. Недостаток этого и предыдущего способов в том, что в исходный текст программы включаются абсолютно все макроопределения. Для исправления ситуации можно использовать директиву *purge*, в качестве операндов которой

через запятую перечисляются имена макрокоманд, которые не должны включаться в текст программы. К примеру:

```
...  
include    iomac.inc  
purge      outstr,exit
```

...

В данном случае в исходный текст программы перед началом компиляции TASM вместо строки `include iomac.inc` вставит строки из файла `iomac.inc`. Но вставленный текст будет отличаться от оригинала тем, что в нем будут отсутствовать макроопределения `outstr` и `exit`.

А теперь вернемся к программе из листинга 3.1. Проанализируем ее текст, выявим повторяющиеся участки и составим для них макроопределения (листинг 13.1).

Листинг 13.1. Пример 1 создания и использования макрокоманд

```
<1>      :prg_3_1.asm с макроопределениями  
<2>      init_ds    macro  
<3>      ;Макрос настройки ds на сегмент данных  
<4>          mov     ax,data  
<5>          mov     ds,ax  
<6>          endm  
<7>      out_str   macro str  
<8>      ;Макрос вывода строки на экран.  
<9>      ;На входе – выводимая строка.  
<10>     ;На выходе – сообщение на экране.  
<11>          push    ax  
<12>          mov     ah,09h  
<13>          mov     dx,offset str  
<14>          int     21h  
<15>          pop     ax  
<16>          endm  
<17>  
<18>      clear_r   macro rg  
<19>      ;очистка регистра rg  
<20>          xor     rg,rg  
<21>          endm  
<22>  
<23>      get_char  macro  
<24>      ;ввод символа  
<25>      ;введенный символ в а1  
<26>          mov     ah,1h  
<27>          int     21h  
<28>          endm  
<29>  
<30>      conv_16_2  macro  
<31>      ;макрос преобразования символа шестнадцатеричной цифры  
<32>      ;в ее двоичный эквивалент в а1  
<33>          sub    d1,30h  
<34>          cmp    d1,9h  
<35>          jle    $+5
```

```

<36>           sub    d1,7h
<37>           endm
<38>
<39> exit macro
<40> ;макрос конца программы
<41>     mov    ax,4C00h
<42>     int    21h
<43>     endm
<44>
<45> data segment para public "data"
<46> message db      "Введите две шестнадцатеричные цифры (буквы
              A,B,C,D,E,F – прописные): $"
<47> data ends
<48>
<49> stk  segment stack
<50> db    256 dup(?)
<51> stk  ends
<52>
<53> code segment para public "code"
<54> assume cs:code,ds:data,ss:stk
<55> main proc
<56>   init_ds
<57>   out_str    message
<58>   clear_r    ax
<59>   get_char
<60>   mov    d1,a1
<61>   conv_16_2
<62>   mov    c1,4h
<63>   shl    d1,c1
<64>   get_char
<65>   conv_16_2
<66>   add    d1,a1
<67>   xchg  d1,a1 :результат в a1
<68>   exit
<69> main endp
<70> code ends
<71> end  main

```

В листинге 13.1 в строках 2–7, 8–16, 18–21, 23–28, 30–37, 39–43 описаны макроопределения. Их назначение приведено сразу после заголовка в теле каждого макроопределения. Все эти макроопределения можно использовать и при написании других программ. Посмотрите на модернизированный исходный текст программы из листинга 3.1 в листинге 13.1 (строки 55–69). Если не обращать внимания на некоторые неясные моменты, то сам сегмент кода стал внешне более читабельным, и даже можно сказать, что в нем появился какой то смысл. Откомпилируйте листинг 13.1 и получите файл листинга. После этого сравните исходный текст программы и то, каким он стал после его обработки ассемблером. Вы увидите, что текст программы изменился, — после строк программы, в которых были макрокоманды, появились фрагменты текста. Вид этих фрагментов зависит от того, есть ли у макрокоманды параметры. Уделите анализу файла листинга немного времени.

Функционально макроопределения похожи на процедуры. Сходство их в том, что и те, и другие достаточно один раз где-то описать, а затем вызывать их специальным образом. На этом их сходство заканчивается, и начинаются различия, которые в зависимости от целевой установки можно рассматривать и как достоинства, и как недостатки:

- в отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении менее гибки;
- при каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстродействие несколько снижается за счет необходимости осуществления переходов.

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

имя_макрокоманды список_фактических_аргументов

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

К примеру рассмотрим самое короткое макроопределение в листинге 13.1 — *clear_rg*. Как отмечено выше, результаты работы макроассемблера можно узнать, просмотрев файл листинга после трансляции. Покажем несколько его фрагментов, которые демонстрируют, как был описан текст макроопределения *clear_rg* (строки 24–27), как был осуществлен вызов макрокоманды *clear_rg* с фактическим параметром *ax* (строка 74) и как выглядит результат работы макрогенератора, сформировавшего команду ассемблера *xor ax,ax* (строка 75):

```
24      clear_r    macro rg
25      ;очистка регистра rg
26      xor    rg,rg
27      endm
...
...
```

74 clear_r ax
75 000E 33 C0 xor ax,ax

Таким образом, в итоге мы получили то, что и требовалось, — команду очистки заданного регистра, в данном случае ax. В другом месте программы вы можете выдать ту же макрокоманду, но уже с другим именем регистра. Если у вас есть желание, то вы можете провести эксперименты с этой и другими макрокомандами. Каждый фактический аргумент представляет собой строку символов, для формирования которой применяются следующие правила:

○ строка может состоять:

- из последовательности символов без пробелов, точек, запятых, точек с запятой;
- из последовательности любых символов, заключенных в угловые скобки: <...>. В этой последовательности можно указывать как пробелы, так и точки, запятые, точки с запятыми. Не забывайте о том, что угловые скобки <> — это тоже оператор ассемблера. Мы упоминали о них при обсуждении директивы equ;

○ для того чтобы указать, что некоторый символ внутри строки, представляющей фактический параметр, является собственно символом, а не чем-то иным, например, некоторым разделителем или ограничивающей скобкой, применяется специальный оператор «!!». Этот оператор ставится непосредственно перед описанным выше символом, и его действие эквивалентно заключению данного символа в угловые скобки (см. предыдущий пункт);

○ если требуется вычисление в строке некоторого константного выражения, то в начале этого выражения нужно поставить знак %:

% *константное выражение* — значение *константное выражение* вычисляется и подставляется в текстовом виде в соответствии с текущей системой счисления¹.

Теперь обсудим вопрос, как транслятор распознает формальные аргументы в теле макроопределения для их последующей замены на фактические аргументы?

Прежде всего, по их именам в заголовке макроопределения. В процессе генерации макрорасширения компилятор ассемблера ищет в тексте тела макроопределения последовательности символов, совпадающие с теми последовательностями символов, из которых состоят формальные параметры. После обнаружения такого совпадения формальный параметр из тела макроопределения замещается соответствующим фактическим параметром из макрокоманды. Этот процесс называется *подстановкой аргументов*. Здесь нужно отметить еще раз особо *список формальных аргументов* в заголовке макроопределения. В общем случае он содержит не только

¹ Под текущей системой счисления понимается то, как интерпретируются транслятором числа или строки с фиксированным числовым значением — как двоичные, десятичные или шестнадцатеричные числа. По умолчанию транслятор трактует их как десятичные. Ассемблер имеет специальную директиву .radix, которая дает возможность изменить текущую систему счисления. В качестве операнда директива .radix имеет значение 2, 10 или 16, что означает выбор, соответственно, двоичной, десятичной или шестнадцатеричной системы счисления.

перечисление формальных элементов через запятую, но и некоторую дополнительную информацию. Полный синтаксис формального аргумента следующий:

имя_формального_аргумента[*:тип*],

где *тип* может принимать значения:

- REQ, которое говорит о том, что требуется обязательное явное задание фактического аргумента при вызове макрокоманды;
- =<любая_строка> — если аргумент при вызове макрокоманды не задан, то в соответствующие места в макрорасширении будет вставлено значение по умолчанию, соответствующее значению *любая_строка*. Будьте внимательны: символы, входящие в *любая_строка*, должны быть заключены в угловые скобки.

Но не всегда ассемблер может распознать в теле макроопределения формальный аргумент. Это, например, может произойти в случае, когда он является частью некоторого идентификатора. В этом случае последовательность символов формального аргумента отделяют от остального контекста с помощью специального символа &. Этот прием часто используется для задания модифицируемых идентификаторов и кодов операций. К примеру, определим макрос, который предназначен для генерации в программе некоторой таблицы, причем параметры этой таблицы можно задавать с помощью аргументов макрокоманды:

```
...
def_table macro type=b,len=REQ
tabl_&type      d&type len dup (0)
endm
...
.data
def_tabl b,10
def_tabl w,5
```

После того как вы подвергнете трансляции текст программы, содержащий эти строки, вы получите следующие макрорасширения:

```
tabl_b    db    10 dup (0)
tabl_w    dw    10 dup (0)
```

Символ & можно применять и для распознавания формального аргумента в строке, заключенной в кавычки ". Например:

```
num_char macro message
;...
;подсчитать количество (num) символов в строке
    jmp    m1
elem    db    "Строка &message содержит"
;число символов в строке message в коде ASCII
numdb    2 dup (0)
    db    " символов",10,13,'$'      ;конец строки
                                ;для вывода функцией 09h
m1:
;...
;вывести elem на экран
endm
```

В связи с рассмотрением последнего фрагмента разберем ситуацию, когда тело макроопределения содержит метку или имя в директиве резервирования и инициализации данных. Если в программе некоторая макрокоманда вызывается несколько раз, то в процессе макрогенерации возникнет ситуация, когда в программе один идентификатор будет определен несколько раз, что, естественно, будет распознано транслятором как ошибка. Для выхода из подобной ситуации применяют директиву `local`, которая имеет следующий синтаксис:

`local список_идентификаторов`

Эту директиву необходимо задавать непосредственно за заголовком макропределения. Результатом работы этой директивы будет генерация в каждом экземпляре макрорасширения уникальных имен для всех идентификаторов, перечисленных в `список_идентификаторов`. Эти уникальные имена имеют вид `??xxxx`, где `xxxx` — шестнадцатеричное число. Для первого идентификатора в первом экземпляре макрорасширения `xxxx= 0000`, для второго — `xxxx= 0001` и т. д. Контроль за правильностью размещения и использования этих уникальных имен берет на себя ассемблер. Для того чтобы вам окончательно все стало понятно, введем и оттранслируем листинг 13.2. В нем, кроме некоторых ранее рассмотренных макрокоманд, содержится макрокоманда `num_char`. Ее назначение — подсчитывать количество символов в строке, адрес которой передается этой макрокоманде в качестве фактического параметра. Стока должна удовлетворять требованию, предъявляемому к строке, предназначеннной для вывода на экран функцией `09h` прерывания `21h`, то есть заканчиваться символом `$`. Другой момент, который нашел отражение в этой программе, — использование символа `&` для распознавания формального аргумента в строке, заключенной в кавычки " " (см. последний фрагмент).

Листинг 13.2. Пример 2 создания и использования макрокоманд

```
:prg_13_2.asm
init_ds    macro
;макрос настройки ds на сегмент данных
    mov     ax,data
    mov     ds,ax
    xor     ax,ax
    endm

out_str   macro      str
;макрос вывода строки на экран.
;На входе — выводимая строка.
;На выходе — сообщение на экране.
    push    ax
    mov     ah,09h
    mov     dx,offset str
    int     21h
    pop     ax
    endm

exit macro
;макрос конца программы
    mov     ax,4c00h
    int     21h
    endm
```

Листинг 13.2 (продолжение)

```

num_char    macro    message
local m1,elem,num,err_mes,find,num_exit
;макрос подсчета количества символов в строке.
;Длина строки – не более 99 символов.
;Вход: message – адрес строки символов, ограниченной "$"
;Выход: в al – количество символов в строке message
;и вывод сообщения
    jmp    m1
elem db    "Строка &message содержит "
num  db    2 dup (0)           ;число символов в строке
                ;message в коде ASCII
db    " символов",10,13,'$' ;конец строки
                ;для вывода функцией 09h
err_mes db    "Строка &message не содержит символа конца строки",10,13,'$'
m1:
;сохраняем используемые в макросе регистры
    push   es
    push   cx
    push   ax
    push   di
    push   ds
    pop    es          ;настройка es на ds
    mov    al,'$'       ;символ для поиска – "$"
    cld
    lea    di,message  ;загрузка в es:di смещения
                        ;строки message
    push   di          ;запомним di – адрес начала строки
    mov    cx,99        ;для префикса repne – максимальная
                        ;длина строки
                        ;поиск в строке (пока нужный символ
                        ;и символ в строке не равны)
                        ;выход – при первом совпадении
repnescasb
    je    find          ;если символ найден – переход на обработку
;вывод сообщения о том, что символ не найден
    push   ds
;подставляем cs вместо ds для функции 09h (int21h)
    push   cs
    pop    ds
    out_str err_mes
    pop    ds
    jmp    num_exit     ;выход из макроса
;совпали
;считаем количество символов в строке:
find:
    pop    ax          ;восстановим адрес начала строки
    sub    di,ax         ;(di)=(di)-(ax)
    xchg  di,ax         ;(di) <-> (ax)
    sub    al,3          ;корректировка на служебные
                        ;символы – 10, 13, "$"
    aam
    or    ax,3030h       ;в al две упакованные BCD-цифры
                        ;результата подсчета
    ;преобразование результата в код ASCII

```

```

    mov    cs:num.ah
    mov    cs:num+1.al
;вывести elem на экран
    push   ds
;подставляем cs вместо ds для функции 09h (int21h)
    push   cs
    pop    ds
    out_str elem
    pop    ds
num_exit:
    push   di
    push   ax
    push   cx
    push   es
    endm
data    segment para public "data"
msg_1   db    "Строка_1 для испытания",10,13,'$'
msg_2   db    "Строка_2 для второго испытания",10,13,'$'
data    ends
stksegment stack
    db    256 dup(?)
stkends
code    segment para public "code"
assume cs:code,ds:data,ss:stk
main    proc
    init_ds
    out_str msg_1
    num_char msg_1
    out_str msg_2
    num_char msg_2
    exit
main    endp
code    ends
end main

```

В теле макроопределения можно размещать *комментарии* и делать это особым образом. Если применить для обозначения комментария не одну, как обычно, а две подряд идущие точки с запятой, то при генерации макрорасширения этот комментарий будет исключен. Если по какой-то причине необходимо присутствие комментария в макрорасширении, то его нужно задавать обычным образом, то есть с помощью одинарной точки с запятой. Например:

```

mesmacro message
...;этот комментарий будет включен в текст листинга
...;;этот комментарий не будет включен в текст листинга
endm

```

Макродирективы

С помощью макросредств ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и

даже порядок их следования. Сделать это можно с помощью набора *макродиректив* (далее — просто директив). Их можно разделить на две группы:

- директивы *повторения WHILE, REPT, IRP и IRPC*. Директивы этой группы предназначены для создания макросов, содержащих несколько идущих подряд одинаковых последовательностей строк. При этом возможна частичная модификация этих строк;
- директивы *управления процессом генерации* макрорасширения EXITM и GOTO. Они предназначены для управления процессом формирования макрорасширения из набора строк соответствующего макроопределения. С помощью этих директив можно как исключать отдельные строки из макрорасширения, так и вовсе прекращать процесс генерации. Директивы EXITM и GOTO обычно используются вместе с условными директивами компиляции, поэтому они будут рассмотрены вместе с ними.

Директивы WHILE и REPT

Директивы WHILE и REPT применяют для повторения определенное количество раз некоторой последовательности строк. Эти директивы имеют следующий синтаксис:

```
WHILE константое_выражение  
последовательность_строк  
ENDM
```

```
REPT константое_выражение  
последовательность строк  
ENDM
```

Обратите внимание, что последовательность повторяемых строк в обеих директивах ограничена директивой ENDM.

При использовании директивы WHILE макрогенератор транслятора будет повторять *последовательность_строк* до тех пор, пока значение *константое_выражение* не станет равно нулю. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (то есть значение *константое_выражение* должно подвергаться изменению внутри *последовательность_строк* в процессе макрогенерации).

Директива REPT, подобно директиве WHILE, повторяет *последовательность_строк* столько раз, сколько это определено значением *константое_выражение*. Отличие этой директивы от WHILE состоит в том, что она автоматически уменьшает на единицу значение *константое_выражение* после каждой итерации. В качестве примера рассмотрим листинг 13.3. В нем демонстрируется применение директив WHILE и REPT для резервирования области памяти в сегменте данных. Имя идентификатора и длина области задаются в качестве параметров для соответствующих макросов def_sto_1 и def_sto_2.

Заметьте, что счетчик повторений в директиве REPT уменьшается автоматически после каждой итерации цикла. Проанализируйте результат трансляции листинга 13.3.

Листинг 13.3. Использование директив повторения

```
;prg_13_3.asm
def_sto_1 macro id_table,len:=<5>
;макрос резервирования памяти длиной len.
;Используется WHILE
id_table label byte
len=len
    while len
        db 0
        len=len-1
    endm
endm
def_sto_2 macro id_table,len
;макрос резервирования памяти длиной len
id_table label byte
    rept len
        db 0
    endm
endm

data      segment para public "data"
    def_sto_1    tab_1,10
    def_sto_2    tab_2,10
data      ends
;сегменты данных и стека в этой программе необязательны
end
```

Таким образом, директивы REPT и WHILE удобно применять для «размножения» в тексте программы последовательности одинаковых строк без внесения в эти строки каких-либо изменений. Следующие две директивы, IRP и IRPC, делают этот процесс более гибким, позволяя модифицировать на каждой итерации некоторые элементы в последовательность_строк.

Директива IRP

Директива IRP имеет следующий синтаксис:

```
IRP формальный_аргумент,<строка_символов_1, ..., строка_символов_n>
последовательность_строк
ENDM
```

Действие данной директивы заключается в том, что она повторяет последовательность_строк n раз, то есть столько раз, сколько строк_символов заключено в угловые скобки во втором операнде директивы IRP. Но это еще не все. Повторение последовательности_строк сопровождается заменой в ней формального_аргумента строкой символов из второго операнда. Так, при первой генерации последовательности_строк формальный_аргумент в них заменяется на строка_символов_1. Если есть строка_символов_2, то это приводит к генерации второй копии последовательность_строк, в которой формальный_аргумент заменяется на строка_символов_2. Эти действия продолжаются до строка_символов_n включительно.

К примеру рассмотрим результат определения в программе следующей конструкции:

```
irp    ini,<1,2,3,4,5>
db    ini
endm
```

Макрогенератором будет сгенерировано следующее макрорасширение:

```
db    1
db    2
db    3
db    4
db    5
```

Директива IRPC

Директива IRPC имеет следующий синтаксис:

IRPC формальный_аргумент,строка_символов

последовательность строк

ENDM

Действие данной директивы подобно IRP, но отличается тем, что она на каждой очередной итерации заменяет формальный_аргумент очередным символом из строка_символов. Понятно, что количество повторений последовательность_строк будет определяться количеством символов в строке_символов. К примеру:

```
irpc    rg,<abcd>
push    rg&x
endm
```

В процессе макрогенерации эта директива развернется в следующую последовательность строк:

```
push    ax
push    bx
push    cx
push    dx
```

Директивы условной компиляции

Последний тип макросредств — директивы *условной компиляции*. Существует два типа этих директив:

- *директивы компиляции по условию* позволяют проанализировать определенные условия в ходе генерации макрорасширения и при необходимости изменить этот процесс;
- *директивы генерации ошибок по условию* также контролируют ход генерации макрорасширения с целью генерации или обнаружения определенных ситуаций, которые могут интерпретироваться как ошибочные.

С этими директивами применяются директивы управления процессом генерации макрорасширений EXITM и GOTO.

Директива EXITM не имеет операндов, и ее действие заключается в том, что она немедленно прекращает процесс генерации макрорасширения, начиная с того места, где она встретилась в макроопределении.

Директива GOTO *имя_метки* переводит процесс генерации макроопределения в другое место, прекращая тем самым последовательное разворачивание строк макроопределения. Метка, на которую передается управление, имеет специальный формат:

:*имя_метки*

Примеры применения этих директив будут приведены ниже.

Директивы компиляции по условию

Данные директивы предназначены для организации выборочной трансляции фрагментов программного кода. Такая выборочная компиляция означает, что в макрорасширение включаются не все строки макроопределения, а только те, которые удовлетворяют определенным условиям. То, какие конкретно условия должны быть проверены, определяется типом условной директивы. Введение в язык ассемблера этих директив значительно повышает его мощь. Всего имеются 10 типов условных директив компиляции. Их логично попарно объединить в четыре группы:

- директивы IF и IFE — условная трансляция по результату вычисления логического выражения;
- директивы IFDEF и IFNDEF — условная трансляция по факту определения символьического имени;
- директивы IFB и IFNB — условная трансляция по факту определения фактического аргумента при вызове макрокоманды;
- директивы IFIDN, IFIDNI, IFDIF и IFDIFI — условная трансляция по результату сравнения строк символов.

Условные директивы компиляции имеют общий синтаксис и применяются в составе следующей синтаксической конструкции:

```
IFxxx      логическое_выражение_или_аргументы
фрагмент_программы_1
    ELSE
        фрагмент_программы_2
ENDIF
```

Заключение некоторых фрагментов текста программы — *фрагмент_программы_1* и *фрагмент_программы_2* — между директивами IFxxx, ELSE и ENDIF приводит к их выборочному включению в объектный модуль. Какой именно из этих фрагментов — *фрагмент_программы_1* или *фрагмент_программы_2* — будет включен в объектный модуль, зависит от конкретного типа условной директивы, задаваемого значением *xxx*, и значения условия, определяемого операндом (операндами) условной директивы *логическое_выражение_или_аргумент(ы)*.

Директивы IF и IFE

Синтаксис этих директив следующий:

IF(E) логическое_выражение

фрагмент_программы_1

ELSE

фрагмент_программы_2

ENDIF

Обработка этих директив макроассемблером заключается в вычислении логического выражения и включении в объектный модуль *фрагмент_программы_1* или *фрагмент_программы_2* в зависимости от того, в какой директиве – IF или IFE – это выражение встретилось:

- если в директиве IF логическое выражение истинно, то в объектный модуль помещается *фрагмент_программы_1*. Если логическое выражение ложно, то при наличии директивы ELSE в объектный код помещается *фрагмент_программы_2*. Если же директивы ELSE нет, то вся часть программы между директивами IF и ENDIF игнорируется и в объектный модуль ничего не включается. Кстати сказать, понятие истинности и ложности значения логического выражения весьма условно. Ложным оно будет считаться, если его значение равно нулю, а истинным – при любом значении, отличном от нуля.
- директива IFE, аналогично директиве IF, анализирует значение логического выражения. Но теперь для включения *фрагмент_программы_1* в объектный модуль требуется, чтобы *логическое_выражение* имело значение «ложь».

Директивы IF и IFE очень удобно использовать при необходимости изменения текста программы в зависимости от некоторых условий. К примеру составим макрос для определения в программе области памяти длиной не более 50 и не менее 10 байт (листинг 13.4).

Листинг 13.4. Использование условных директив IF и IFE

```
<1>      :prg_13_4.asm
<2>      masm
<3>      model small
<4>      stack 256
<5>      def_tab_50 macro len
<6>      if    len GE 50
<7>      GOTO exit
<8>      endif
<9>      if    len LT 10
<10>     :exit
<11>     EXITM
<12>     endif
<13>     rept len
<14>           db    0
<15>     endm
<16>     endm
<17>     .data
```

```
<18>    def_tab_50 15
<19>    def_tab_50 5
<20>    .code
<21>    main:
<22>        mov    ax,@data
<23>        mov    ds,ax
<24>        exit:
<25>        mov    ax,4c00h
<26>        int    21h
<27>    end   main
```

Ведите и оттранслируйте листинг 13.4. Не забывайте о том, что условные директивы действуют только на шаге трансляции, и поэтому результат их работы можно увидеть лишь после макрогенерации, то есть в листинге программы. В нем вы увидите, что в результате трансляции строка 18 листинга 13.3 развернется в пятнадцать нулевых байт, а строка 19 оставит макрогенератор совершенно равнодушным, так как значение фактического операнда в строках 6 и 9 будет ложным. Обратите внимание, что для обработки реакции на ложный результат анализа в условной директиве мы использовали макродирективы EXITM и GOTO. Наверное, в данном случае можно было составить и более оптимальный вариант макрокоманды для резервирования некоторого пространства памяти в сегменте данных, но такой способ был выбран, исходя из учебных целей.

Другой интересный и полезный вариант применения директив IF и IFE – отладочная печать. Суть здесь в том, что в процессе отладки программы почти всегда возникает необходимость динамически отслеживать состояние определенных программно-аппаратных объектов, в качестве которых могут выступать переменные, регистры микропроцессора и т. п. После этапа отладки отпадает необходимость в таких диагностических сообщениях. Для их устранения нужно корректировать исходный текст программы, после чего ее следует подвергнуть повторной трансляции. Но есть более изящный выход. Можно определить в программе некоторую переменную, к примеру debug, и использовать ее совместно с условными директивами IF или IFE. К примеру:

```
<1>    ...
<2>    debug equ    1
<3>    ...
<4>    .code
<5>    ...
<6>    if    debug
<7>        ;любые команды и директивы ассемблера
<8>        ;(вывод на печать или монитор)
<9>    endif
```

На время отладки и тестирования программы вы можете заключить отдельные участки кода в своеобразные операторные скобки в виде директив IF и ENDIF (строки 6–9 последнего фрагмента), которые реагируют на значение логической переменной debug. При значении debug = 0 транслятор полностью проигнорирует текст внутри этих условных операторных скобок; при debug = 1, наоборот, будут выполнены все действия, описанные внутри них.

Директивы IFDEF и IFNDEF

Синтаксис этих директив следующий:

```
IF(N)DEF символическое_имя
фрагмент_программы_1
ELSE
фрагмент_программы_2
ENDIF
```

Данные директивы позволяют управлять трансляцией фрагментов программы в зависимости от того, определено или нет в программе некоторое *символическое_имя*. Директива IFDEF проверяет, описано или нет в программе *символическое_имя*, и если это так, то в объектный модуль помещается *фрагмент_программы_1*. В противном случае, при наличии директивы ELSE, в объектный код помещается *фрагмент_программы_2*. Если же директивы ELSE нет (и *символическое_имя* в программе не описано), то вся часть программы между директивами IF и ENDIF игнорируется и в объектный модуль не включается.

Действие IFNDEF обратно IFDEF. Если символического имени в программе нет, то транслируется *фрагмент_программы_1*. Если оно присутствует, то при наличии ELSE транслируется *фрагмент_программы_2*. Если ELSE отсутствует, а *символическое_имя* в программе определено, то часть программы, заключенная между IFNDEF и ENDIF, игнорируется.

В качестве примера рассмотрим ситуацию, когда в объектный модуль программы должен быть включен один из трех фрагментов кода. Какой из трех фрагментов будет включен в объектный модуль, зависит от значения некоторого идентификатора switch:

- если switch = 0, то сгенерировать фрагмент для вычисления выражения $y = x^2 * n$;
- если switch = 1, то сгенерировать фрагмент для вычисления выражения $y = x / 2 * n$;
- если switch не определен, то ничего не генерировать.

Соответствующий фрагмент исходной программы может выглядеть так:

```
ifndef SW          ;если SW не определено, то выйти из макроса
EXITM
else              ;иначе – на вычисление
    mov cl,n
    ife SW
        sal x,cl ;умножение на степень 2
        ;сдвигом влево
    else
        sar x,cl ;деление на степень 2
        ;сдвигом вправо
    endif
endif
```

Как видим, эти директивы логически связаны с директивами IF и IFE, то есть их можно применять в тех же самых случаях, что и последние. Есть еще одна интересная возможность использования этих директив. На уроке 4 мы обсуж-

дали формат командной строки TASM и говорили об опциях, которые можно в ней задавать. Вспомните одну из них — опцию /d *идентификатор_значение*. Ее использование дает возможность управлять значением идентификатора прямо из командной строки, не изменяя при этом текста программы. В качестве примера рассмотрим листинг 13.5. В этом примере мы пытаемся с помощью макроса контролировать процесс резервирования и инициализации некоторой области памяти в сегменте данных.

Листинг 13.5. Инициализация значения идентификатора из командной строки

```
<1>      :prg_13_5.asm
<2>      masm
<3>      model small
<4>      stack 256
<5>      def_tab_50 macro len
<6>      ifndef len
<7>          display      "size_m не определено, задайте значение 10<size_m<50"
<8>          exitm
<9>      else
<10>         if    len GE 50
<11>             GOTO exit
<12>         endif
<13>         if    len LT 10
<14>             :exit
<15>             EXITM
<16>         endif
<17>         rept   len
<18>             db    0
<19>         endm
<20>     endif
<21>     endm
<22>     :size_m=15
<23>     .data
<24>     def_tab_50 size_m
<25>
<26>     .code
<27>     main:
<28>         mov    ax,@data
<29>         mov    ds,ax
<30>         exit:
<31>         mov    ax,4c00h
<32>         int    21h
<33>     end main
```

Запустив этот пример на трансляцию, вы получите сообщение о том, что забыли определить значение переменной *size_m*. Исправить эту ошибку можно одним из двух приведенных ниже действий:

- определите где-то в начале исходного текста программы значение этой переменной с помощью *equ*:
size_m equ 15
- запустите программу на трансляцию командной строкой вида:
tasm /dsize_m=15 /zi prg_13_2...

В листинге 13.5 мы использовали еще одну возможность транслятора — директиву `display`, с помощью которой можно формировать пользовательское сообщение в процессе трансляции программы. Директива `display` будет рассмотрена в конце данного урока.

Директивы IFB и IFNB

Синтаксис этих директив следующий:

```
IF(N)B    аргумент
фрагмент_программы_1
ELSE
фрагмент_программы_2
ENDIF
```

Данные директивы используются для проверки фактических параметров, передаваемых в макрос. При вызове макрокоманды они анализируют значение аргумента и в зависимости от того, равно оно пробелу или нет, транслируется либо *фрагмент_программы_1*, либо *фрагмент_программы_2*. Какой именно фрагмент будет выбран, зависит от кода директивы:

- Директива IFB проверяет равенство аргумента пробелу. В качестве аргумента могут выступать имя или число. Если его значение равно пробелу (то есть фактический аргумент при вызове макрокоманды не был задан), то транслируется и помещается в объектный модуль *фрагмент_программы_1*. В противном случае, при наличии директивы ELSE, в объектный код помещается *фрагмент_программы_2*. Если же директивы ELSE нет, то при равенстве аргумента пробелу вся часть программы между директивами IFB и ENDIF игнорируется и в объектный модуль не включается.
- Действие IFNB обратно IFB. Если значение аргумента в программе не равно пробелу, то транслируется *фрагмент_программы_1*. В противном случае, при наличии директивы ELSE, в объектный код помещается *фрагмент_программы_2*. Если же директивы ELSE нет, то вся часть программы (при неравенстве аргумента пробелу) между директивами IFNB и ENDIF игнорируется и в объектный модуль не включается.

В качестве типичного примера применения этих директив предусмотрим строки в макроопределении, которые будут проверять, указывается ли фактический аргумент при вызове соответствующей макрокоманды:

```
show    macro reg
ifb<reg>
display "не задан регистр"
exitm
endif
...
endm
```

Если теперь в сегменте кода вызвать макрос `show` без аргументов, то будет выведено сообщение о том, что не задан регистр, и генерация макрорасширения будет прекращена директивой `exitm`.

Директивы IFIDN, IFIDNI, IFDIF и IFDIFI

Эти директивы позволяют не просто проверить наличие или значение аргументов макрокоманды, но и выполнить идентификацию аргументов как строк символов.

Синтаксис этих директив:

IFIDN(I) аргумент_1,аргумент_2

фрагмент_программы_1

ELSE

фрагмент_программы_2

ENDIF

IFDIF(I) аргумент_1,аргумент_2

фрагмент_программы_1

ELSE

фрагмент_программы_2

ENDIF

В этих директивах проверяются *аргумент_1* и *аргумент_2* как строки символов. Каждой именно код — *фрагмент_программы_1* или *фрагмент_программы_2* — будет транслироваться по результатам сравнения, зависит от кода директивы. Парность этих директив объясняется тем, что они позволяют учитывать либо не учитывать различие строчных и прописных букв. Так, директивы IFIDNI и IFDIFI игнорируют это различие, а IFIDN и IFDIF — учитывают.

Директива IFIDN(I) сравнивает символьные значения *аргумент_1* и *аргумент_2*. Если результат сравнения положительный, то *фрагмент_программы_1* транслируется и помещается в объектный модуль. В противном случае, при наличии директивы ELSE, в объектный код помещается *фрагмент_программы_2*. Если же директивы ELSE нет, то вся часть программы между директивами IFIDN(I) и ENDIF игнорируется и в объектный модуль не включается.

Действие IFDIF(I) обратно IFIDN(I). Если результат сравнения отрицательный (строки не совпадают), транслируется *фрагмент_программы_1*. В противном случае все происходит аналогично выше рассмотренным директивам.

Как мы уже упоминали выше, эти директивы удобно применять для проверки фактических аргументов макрокоманд. К примеру проверим, какой из регистров — *a1* или *ah* — передан в макрос в качестве параметра (проверка проводится без учета различия строчных и прописных букв):

```
show      macro rg
ifdifi   <a1>,<rg>
goto     M_a1
else
endiffi   <ah>,<rg>
goto     M_ah
else
exitm
endif
endiff
:M_a1
...
```

```
:M_ah  
...  
endm
```

Вложенность директив условной трансляции

Как мы неоднократно видели в приведенных выше примерах, TASM допускает вложенность условных директив компиляции. Более того, так как вложенность требуется довольно часто, TASM предоставляет набор дополнительных директив формата ELSEIFxxx, которые заменяют последовательность подряд идущих ELSE и IFxxx в структуре:

```
IFxxx  
:...  
    ELSE  
    IFxxx  
;...  
ENDIF  
ENDIF
```

Эту последовательность условных директив можно заменить эквивалентной последовательностью дополнительных директив:

```
IFxxx  
:...  
    ELSEIFxxx  
;...  
ENDIF
```

Наличие xxx в ELSExxx говорит о том, что каждая из директив — IF, IFB, IFIDN и т. д. — имеет аналогичную директиву ELSEIF, ELSEIFB, ELSEIFIDN и т. д. В конечном итоге это улучшает читаемость кода. В последнем примере фрагмента макроса, проверяющем, имя какого регистра было передано в макрос, наблюдается подобная ситуация. Последовательности ELSE и IFIFI можно записать так, как в строке 4:

```
<1>show macro rg  
<2>ifdfi<a1>,<rg>  
<3>goto M_a1  
<4>elseifdfi <ah>,<rg>  
<5>goto M_ah  
<6>else  
<7>exitm  
<8>endif  
<9>:M_a1  
<10> ...  
<11> :M_ah  
<12> ...  
<13> endm
```

Директивы генерации ошибок

В языке TASM есть ряд директив, называемых *директивами генерации пользовательской ошибки*. Их можно рассматривать и как самостоятельное средство, и как

метод, расширяющий возможности директив условной компиляции. Они предназначены для обнаружения различных ошибок в программе, таких как неопределенные метки или пропуск параметров макроса. Директивы генерации пользовательской ошибки по принципу работы можно разделить на два типа:

- безусловные директивы, генерирующие ошибку трансляции без проверки каких-либо условий;
- условные директивы, генерирующие ошибку трансляции после проверки определенных условий.

Большинство директив генерации ошибок имеет два обозначения, хотя принцип их работы одинаков. Второе название отражает их сходство с директивами условной компиляции. При дальнейшем обсуждении такие парные директивы будут приводиться в скобках.

Безусловная генерация пользовательской ошибки

К безусловным директивам генерации пользовательской ошибки относится только одна директива — это **ERR (.ERR)**.

Данная директива, будучи вставлена в текст программы, безусловно приводит к генерации ошибки на этапе трансляции и удалению объектного модуля. Она очень эффективна при ее использовании с директивами условной компиляции или в теле макрокоманды с целью отладки. К примеру, эту директиву можно было бы вставить в ту ветвь программы (в последнем рассмотренном нами макроопределении), которая выполняется, если указанный в качестве аргумента регистр отличен от **a1** и **ah**:

```
show    macro rg .
ifdifi  <a1>,<rg>
goto    M_a1
else
ifdifi  <ah>,<rg>
goto    M_ah
else
.Err
endif
endif
...
endm
```

Если после определенного таким образом макроопределения в сегменте кода вызвать макрокоманду **show** с фактическим параметром, отличным от имен регистров **ah** или **a1**, будет сгенерирована ошибка компиляции (с текстом «*User error*»), сам процесс компиляции прекращен и, естественно, объектный модуль создан не будет.

Остальные директивы являются *условными*, так как их поведение определяют некоторые условия.

Условная генерация пользовательской ошибки

Набор условий, на которые реагируют директивы условной генерации пользовательской ошибки, такой же, как и у директив условной компиляции. Поэтому и

количество этих директив такое же. Принцип их работы ясен, поэтому рассматривать их мы будем очень кратко. Заметим только, что как и директивы условной компиляции, использовать большинство директив условной генерации пользовательской ошибки можно как в макроопределениях, так и в любом месте программы.

Директивы .ERRB (ERRIFB) и .ERRNB (ERRIFNB)

Синтаксис директив:

- .ERRB (ERRIFB) <имя_формального_аргумента> — генерация пользовательской ошибки, если <имя_формального_аргумента> пропущено;
- .ERRNB (ERRIFNB) <имя_формального_аргумента> — генерация пользовательской ошибки, если <имя_формального_аргумента> присутствует.

Данные директивы применяются для генерации ошибки трансляции в зависимости от того, задан или нет при вызове макрокоманды фактический аргумент, соответствующий формальному аргументу в заголовке макроопределения с именем <имя_формального_аргумента>. По принципу действия эти директивы полностью аналогичны соответствующим директивам условной компиляции IFB и IFNB. Их обычно используют для проверки задания параметров при вызове макроса.

Строка *имя_формального_аргумента* должна быть заключена в угловые скобки.

К примеру определим обязательность задания фактического аргумента, соответствующего формальному аргументу rg, в макросе show:

```
<1>      show macro rg
<2>      ;если rg в макрокоманде не будет задан,
<3>      ;то завершить компиляцию
<4>      .errb <rg>
<5>      ;текст макроопределения
<6>      ;...
<7>      endm
```

Директивы .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF)

Синтаксис директив:

- .ERRDEF (ERRIFDEF) *символическое_имя* — если указанное имя определено до выдачи этой директивы в программе, то генерируется пользовательская ошибка;
- .ERRNDEF (ERRIFNDEF) *символическое_имя* — если указанное *символическое_имя* не определено до момента обработки транслятором данной директивы, то генерируется пользовательская ошибка.

Данные директивы генерируют ошибку трансляции в зависимости от того, определено или нет некоторое *символическое_имя* в программе. Не забывайте о том, что компилятор TASM по умолчанию формирует объектный модуль за один проход исходного текста программы. Следовательно, директивы .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF) отслеживают факт определения *символического_имени* только в той части исходного текста, которая находится до этих директив.

Директивы .ERRDIF (ERRIFDIF) и .ERRIDN (ERRIFIDN)

Синтаксис директив:

.ERRDIF (ERRIFDIF) <строка_1>, <строка_2> — директива, генерирующая пользовательскую ошибку, если две строки посимвольно не совпадают. Строки могут быть символическими именами, числами или выражениями и должны быть заключены в угловые скобки. Аналогично директиве условной компиляции IFDIF, при сравнении учитывается различие прописных и строчных букв.

.ERRIDN (ERRIFIDN) <строка_1>, <строка_2> — директива, генерирующая пользовательскую ошибку, если строки посимвольно идентичны. Строчное и прописное написание одной и той же буквы воспринимаются как разные символы.

Для того чтобы игнорировать различия строчных и прописных букв, существуют аналогичные директивы:

ERRIFDIFI <строка_1>, <строка_2> — то же, что и ERRIFDIF, но игнорируется различие строчных и прописных букв при сравнении <строка_1> и <строка_2>.

ERRIFIDNI <строка_1>, <строка_2> — то же, что и ERRIFIDN, но игнорируется различие строчных и прописных букв при сравнении <строка_1> и <строка_2>.

Данные директивы, как и соответствующие им директивы условной компиляции, удобно применять для проверки передаваемых в макрос фактических параметров.

Директивы .ERRE (ERRIFE) и .ERRNZ (ERRIF)

Синтаксис директив:

.ERRE (ERRIFE) *константное выражение* — директива вызывает пользовательскую ошибку, если *константное выражение* ложно (равно нулю). Вычисление *константного выражения* должно приводить к абсолютному значению, и это выражение не может содержать компонентов, являющихся ссылками вперед.

.ERRNZ(ERRIF) *константное выражение* — директива вызывает пользовательскую ошибку, если *константное выражение* истинно (не равно нулю). Вычисление *константного выражения* должно приводить к абсолютному значению и не может содержать компонентов, являющихся ссылками вперед.

Константные выражения в условных директивах

Как вы успели заметить, во многих условных директивах в формировании условия участвуют выражения. Результат вычисления этого выражения обязательно должен быть константой. Хотя его компонентами могут быть и символические

параметры, но их сочетание в выражении должно давать абсолютный результат.

К примеру:

```
...
.data
masdb    ...
lendd    ...
...
.code
...
.erre     (len-mas) lt 10      ;генерация ошибки, если длина
;области mas меньше 10 байт
...
```

Кроме того, выражение не должно содержать компоненты, которые транслятор еще не обработал к тому месту программы, где находится условная директива. Также мы отметили, что логические результаты «истина» и «ложь» являются условными в том смысле, что ноль соответствует логическому результату «ложь», а любое ненулевое значение — «истине». Но в языке ассемблера существуют операторы, которые позволяют сформировать и «чисто логический» результат. Это так называемые операторы отношений, выражающие отношение двух значений или константных выражений. В контексте условных директив вместе с операторами отношений можно рассматривать и логические операторы. Результатом работы и тех, и других может быть одно из двух значений:

- **истина** — число, которое содержит двоичные единицы во всех разрядах;
- **ложь** — число, которое содержит двоичные нули во всех разрядах.

Операторы, которые можно применять в выражениях условных директив и которые формируют логические результаты, приведены в табл. 13.1 и 13.2.

Таблица 13.1. Операторы отношений

| Оператор отношения | Синтаксис | Результат |
|---|-------------------------------|--|
| EQ (equal) — равно | выражение_1 EQ выражение_2 | Истина, если выражение_1 равно выражение_2 |
| NE (not equal) — не равно | выражение_1 NE выражение_2 | Истина, если выражение_1 не равно выражение_2 |
| LT (less than) — меньше | выражение_1 LT выражение_2 | Истина, если выражение_1 меньше выражение_2 |
| LE (less or equal) — меньше или равно | выражение_1 LE выражение_2 | Истина, если выражение_1 меньше или равно выражение_2 |
| GT (greater than) — больше | выражение_1 GT выражение_2 | Истина, если выражение_1 больше выражение_2 |
| GE (greater or equal) — больше или равно | выражение_1 GE выражение_2 | Истина, если выражение_1 больше или равно выражение_2 |

Таблица 13.2. Логические операторы

| Логический оператор | Синтаксис | Результат |
|--------------------------------|-----------------------------|--|
| NOT — логическое отрицание | NOT выражение | Истина, если выражение ложно; ложь, если выражение истинно |
| AND — логическое И выражение_1 | AND выражение_2 | Истина, если выражение_1 и выражение_2 истинны |
| OR — логическое ИЛИ | выражение_1 OR выражение_2 | Истина, если выражение_1 или выражение_2 истинны |
| XOR — исключающее ИЛИ | выражение_1 XOR выражение_2 | Истина, если выражение_1 = (NOT выражение_2) |

Дополнительное управление трансляцией

TASM предоставляет средства для вывода текстового сообщения во время трансляции программы — директивы `DISPLAY` и `%OUT`. С их помощью можно при необходимости следить за ходом трансляции. К примеру:

```
display недопустимые аргументы макрокоманды
```

```
...
%out недопустимое имя регистра
```

В результате обработки этих директив на экран будут выведены тексты сообщений. Если эти директивы использовать совместно с директивами условной компиляции, то, к примеру, можно отслеживать путь, по которому осуществляется трансляция исходного текста программы.

В заключение можно предложить читателю уже с этого момента начать формировать набор полезных в его практической работе макрокоманд. В качестве основы вы можете взять файл `mas.inc`, который находится на дискете, прилагаемой к книге, в каталоге данного урока. В дальнейшем, если в этом возникнет необходимость, вы будете самостоятельно дополнять его вашими макросами. Использовать макроопределения из этого файла очень просто, достаточно включить в нужном месте вашей программы строку с директивой `include`, в результате чего в ваш файл будут вставлены строки из файла, указанного в качестве операнда этой директивы.

Основная задача этой книги — научить вас программировать на языке ассемблера. Как вы уже успели понять, нельзя изучать этот язык в отрыве от рассмотрения процессов, происходящих во время выполнения программы на компьютере. Одно из средств изучения таких процессов — отладчик. Но он решает проблему глобально, что далеко не всегда нужно. Тем более, как мы увидим далее, возможности отладчика не безграничны. Поэтому необходимо иметь более универсальное средство, которое бы позволило осуществлять «подглядывание» за содержимым регистра или области памяти динамически, во время выполнения программы. Для этого разработаем еще один макрос. Назовем его, к примеру, `show`. Его аргументом может быть один из четырех регистров — `a1`, `ah`,

ах, eax. С помощью этого макроса можно визуализировать содержимое любого из доступных регистров или области памяти длиной до 32 бит. Для этого достаточно лишь переслать содержимое нужного объекта (регистра или ячейки памяти) с учетом его размера в один из регистров al, ah, ax, eax. Имя одного из этих регистров указывается затем в качестве фактического аргумента макрокоманды show. Второй аргумент этого макроса — позиция на экране. Задавая определенные значения, мы можем судить о том, какая именно макрокоманда show сработала. Еще одна немаловажная особенность данного макроса — в его возможности работать как в реальном, так и защищенном режимах. Распознавание текущего режима работы микропроцессора выполняется автоматически. Текст макроопределения show достаточно велик и по этой причине располагается на прилагаемой к книге дискете в каталоге данного урока. Пример использования этого макроса приведен в листинге 13.6.

Листинг 13.6. Пример использования макроса show

```
;prg_13_6.asm
MASM
MODEL    small
STACK     256
.486p
include  show.inc
.data
pole      dd     3cdf436fh
.code
main:
    mov    ax,@data
    mov    ds,ax
    xor    ax,ax
    mov    ax,1f0fh
    show   a1.0
    show   ah,160
    show   ax,320
    mov    eax,pole
    show   eax,480
exit:
    mov    ax,4c00h
    int    21h
endmain
```

Посвятить время рассмотрению этого макроса полезно еще и потому, что при его разработке было использовано большинство средств, обсуждавшихся на этом уроке.

Подведем некоторые итоги:

- Преимущества языка ассемблера связаны, в частности, с макросредствами. Как говорят, если бы макросредств в нем не было, то их нужно было бы придумать.
- Макросредства — это основные инструменты модификации текста программы на этапе ее трансляции. Принцип работы макросредств основан на прин-

циле препроцессорной обработки, который заключается в том, что текст, поступающий на вход транслятора, перед собственно компиляцией подвергается преобразованию и может значительно отличаться от синтаксически правильного текста, воспринимаемого компилятором. Роль препроцессора в трансляторе TASM выполняет макрогенератор.

- Для того чтобы макрогенератор мог выполнить свою работу, текст программы должен удовлетворять определенным требованиям. Макрогенератору необходимо сообщить, на какие элементы исходного текста он должен реагировать и какие действия должны быть произведены. Можно выделить несколько типов таких элементов.
- Псевдооператоры `equ` и `=` — макрогенератор выполняет простейшие действия, заменяя в последующем тексте программы символическое имя из правой части этих операторов строкой из левой.
- Макрокоманда — строка в исходной программе, которой соответствует специальный блок — макроопределение. Макрокоманда может иметь аргументы, с помощью которых можно изменять текст макроопределения. Макрогенератор, встречая макрокоманду в тексте программы, корректирует текст соответствующего макроопределения, исходя из аргументов этой макрокоманды, и вставляет его в текст программы вместо данной макрокоманды. Процесс такого замещения называется макрогенерацией.
- Условные директивы компиляции позволяют не просто модифицировать отдельные строки программы, но и, исходя из определенных условий, управлять включением в загрузочный модуль отдельных фрагментов программы. Эти директивы наиболее эффективны для организации работы с аргументами, передаваемыми при макрогенерации в макроопределения из макрокоманд, хотя отдельные директивы есть смысл применять и вне макроопределений в любом месте программы.
- Директивы генерации ошибок — подобно условным директивам позволяют анализировать определенные условия в процессе трансляции программы и генерировать ошибку по результатам анализа.

14

УРОК

Модульное программирование

-
- Основы структурного программирования
 - Средства ассемблера для поддержки структурного программирования
 - Процедуры и организация связей между процедурами на языке ассемблера
 - Связь между программами на языках высокого уровня и программами на ассемблере
-

На протяжении всей книги мы неоднократно подчеркивали тот факт, что одним из существенных недостатков программ на языке ассемблера, а значит, и самого языка, является их недостаточная наглядность. По прошествии даже небольшого времени программисту бывает порой трудно разобраться в деталях им же написанной программы. А о чужой программе и говорить не приходится. Если в ней нет хотя бы минимальных комментариев, то разобраться с тем, что она делает, довольно трудно. Причины этого тоже понятны — при программировании на языке ассемблера программисту необходимо производить самые элементарные действия. При этом он должен учитывать и контролировать большое количество информации. Из-за того что производимые операции крайне элементарны, реализовать алгоритм задачи можно по крайней мере несколькими способами. А если способ решения не единственен, то и разобраться в программе подчас бывает нелегко.

По мере накопления опыта эти проблемы частично снимаются. Но одного опыта мало. Ситуация усугубляется, если работа идет в коллективе разработчиков. Тут уже нужны специальные средства. TASM предоставляет следующие организационные и программные средства, позволяющие снять остроту этой проблемы:

- Документирование программистом своей работы и ее результатов. Делается это в первую очередь путем комментирования строк исходного текста программы. При этом комментарии должны коротко, но точно выражать то, что делает данная программа в целом, выделять ее наиболее важные фрагменты и особенности применения отдельных команд. В конечном итоге, комментирование программы облегчает понимание замысла программы, но все-таки полностью не снимает проблему.
- Упрощение кода программы путем замены сложных ее участков более понятным кодом. Для этого, в частности, можно использовать рассмотренный нами механизм макрокоманд.
- Использование при разработке программных проектов достижений современных технологий программирования.

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

Технологии программирования

Последние версии языка ассемблера поддерживают объектно-ориентированное программирование, но реализация его достаточно сложна и требует отдельного

рассмотрения. Типичному процессу написания программы на ассемблере более всего удовлетворяют концепции структурного программирования. Можно даже сказать, что для микропроцессора Intel эти концепции поддерживаются на аппаратном уровне с помощью таких элементов архитектуры, как сегментация памяти и реализация команд передачи управления. На программном уровне поддержка заключается, в основном, в наличии соответствующих средств в конкретном компиляторе. Компилятор TASM имеет все необходимые базовые средства для поддержки структурного программирования. Наш урок будет посвящен рассмотрению этих программно-аппаратных средств.

Структурное программирование

Структурное программирование — методология программирования, базирующаяся на системном подходе к анализу, проектированию и реализации программного обеспечения. Эта методология зародилась в начале 70-х годов и оказалась настолько жизнеспособной, что и до сих пор является основной в большом количестве проектов. Основу этой технологии составляют следующие положения:

- Сложная задача разбивается на более мелкие, функционально лучше управляемые задачи. Каждая задача имеет один вход и один выход. В этом случае управляющий поток программы состоит из совокупности элементарных подзадач с ясным функциональным назначением.
- Простота управляющих структур, используемых в задаче. Это положение означает, что логически задача должна состоять из минимальной, функционально полной совокупности достаточно простых управляющих структур. В качестве примера такой системы можно привести алгебру логики, в которой каждая функция может быть выражена через функционально полную систему: дизъюнкцию, конъюнкцию и отрицание.
- Разработка программы должна вестись поэтапно. На каждом этапе должно решаться ограниченное число четко поставленных задач с ясным пониманием их значения и роли в контексте всей задачи. Если такое понимание не достигается, это говорит о том, что данный этап слишком велик и его нужно разделить на более элементарные шаги.

Понятно, что практическое использование этих положений должно быть подкреплено не только поддержкой со стороны программно-аппаратных средств, но и, в большей степени, личными качествами конкретного программиста, не последними из которых являются наличие опыта проектирования задач и умение подчинить сам процесс программирования некоторой системе.

Конечно, язык ассемблера, в силу его специфики, довольно ограниченно реализует данные положения. В частности, это касается управляющих операторов цикла, условных операторов и операторов выбора. Эти операторы являются встроенными в язык высокого уровня, но в языке ассемблера их попросту нет. Были попытки разработать различные надстройки для языка ассемблера, реализующие эти управляющие операторы, но широкого распространения они не получили. Мне кажется, что это и не нужно, так как реализуемые на языке ассемблера программы решают специфичные системные задачи. Тогда возникает вопрос — а вообще совместимы ли структурное программирование и ассемблер? Ответ

положительный, особенно в том случае, если задача поддается разбиению на более мелкие подзадачи. Тогда реализацию этих подзадач можно осуществить с использованием макрокоманд и процедур. Эти механизмы полностью подходят для реализации так называемого *модульного программирования*, которое является частью структурного подхода. Рассмотрим, что представляет собой модульное программирование.

Концепция модульного программирования

Так же как и для структурной технологии программирования, концепцию модульного программирования можно сформулировать в виде нескольких понятий и положений:

- *Функциональная декомпозиция задачи* – разбиение большой задачи на ряд более мелких, функционально самостоятельных подзадач – *модулей*. Модули связаны между собой только по входным и выходным данным.
- *Модуль* – основа концепции модульного программирования. Каждый модуль в функциональной декомпозиции представляет собой «черный ящик» с одним входом и одним выходом. Модульный подход позволяет безболезненно производить модернизацию программы в процессе ее эксплуатации и облегчает ее сопровождение. Дополнительно модульный подход позволяет разрабатывать части программ одного проекта на разных языках программирования, после чего с помощью компоновочных средств объединять их в единый загрузочный модуль.
- *Реализуемые решения должны быть простыми и ясными*. Если назначение модуля непонятно, то это говорит о том, что декомпозиция начальной или промежуточной задачи была проведена недостаточно качественно. В этом случае необходимо еще раз проанализировать задачу и, возможно, провести дополнительное разбиение на подзадачи. При наличии сложных мест в проекте их нужно подробнее документировать с помощью продуманной системы комментариев. Этот процесс нужно продолжать до тех пор, пока вы действительно не добьетесь ясного понимания назначения всех модулей задачи и их оптимального сочетания.
- Назначение всех переменных модуля должно быть описано с помощью комментариев по мере их определения.
- Исходный текст модуля должен иметь заголовок, в котором отражены как назначение этого модуля, так и все его внешние связи. Этот заголовок можно назвать *интерфейсной частью модуля*. В этой части с использованием комментариев нужно поместить следующую информацию:
 - назначение модуля;
 - особенности функционирования;
 - описание входных аргументов;
 - описание выходных аргументов;
 - использование внешних модулей и переменных;
 - сведения о разработчике для защиты авторских прав.

- В ходе разработки программы следует предусматривать специальные блоки операций, учитывающие реакцию на возможные ошибки в данных или в действиях пользователя. Это очень важный момент, который означает то, что не должно быть тупиковых ветвей в алгоритме программы, в результате работы которых программа «виснет» и перестает отвечать на запросы пользователя. Любые непредусмотренные действия пользователя должны приводить к генерации ошибочной ситуации или к предупреждению о возможности возникновения такой ситуации.

Из этих положений видно, какое большое значение придается организации *управляющих и информационных связей* между структурными единицами программы (модулями), совместно решающими одну или несколько больших задач.

Применительно к языку ассемблера можно рассматривать несколько форм организации управляющих связей:

- Использование механизма макроподстановок, позволяющего изменять исходный текст программы в соответствии с некоторыми предварительно описанными параметризованными объектами. Эти объекты имеют формальные аргументы, что позволяет производить замещение их фактическими аргументами в процессе макрогенерации. Такая форма образования структурных элементов носит некоторый предварительный характер из-за того, что процессы замены происходят на этапе компиляции и есть смысл рассматривать их только как настройку на определенные условия функционирования программы.
- Использование механизма подпрограмм, написанных на ассемблере и структурно входящих в одну программу. В языке ассемблера такие подпрограммы называют *процедурами*. В отличие от макрокоманд, взаимодействие процедур осуществляется на этапе выполнения программы.
- Использование механизма подпрограмм, написанных на разных языках программирования и соединяемых в единый модуль на этапе компоновки. Эта возможность реализуется благодаря унифицированному формату объектного модуля, однозначным соглашениям по передаче аргументов и единым схемам организации памяти на этапе выполнения.
- Использование механизма динамического (то есть времени выполнения) вызова исполняемых модулей и подключения библиотек .dll для операционной системы Windows.

В качестве основных информационных связей можно выделить следующие:

- Использование общих областей памяти и общих программно-аппаратных ресурсов микропроцессора для связи модулей.
- Унифицированную передачу аргументов при *вызове* модуля. Этую унификацию можно представлять двояко: на уровне пользователя и на уровне конкретного компилятора.
- Унифицированную передачу аргументов при *возврате управления из модуля*.

Чуть позже мы подробно рассмотрим процессы, происходящие при передаче аргументов. Сейчас в качестве некоторого итога приведенных выше общих рассуждений перечислим средства языка ассемблера по осуществлению *функциональной декомпозиции* программы:

- макросредства;
- процедуры;
- средства компилятора ассемблера в форме директив организации оперативной памяти и ее сегментации.

Макросредства подробно были рассмотрены на уроке 13. Предварительное обсуждение механизма процедур было проведено при обсуждении команд перехода на уроке 10. Но этот механизм содержит в себе более глубокие вещи, чем тривиальная передача управления из одной точки программы в другую. В частности, он тесно связан со средствами компилятора, поддерживающими организацию памяти и сегментацию. Поэтому дальнейшее наше обсуждение будет посвящено более глубокому изучению функциональной декомпозиции программ с использованием механизма процедур и связанных с ним средств компилятора.

Процедуры в языке ассемблера

В языке ассемблера для оформления процедур как отдельных объектов существуют специальные директивы `PROC/ENDP` и машинная команда `ret`. Эти средства были подробно рассмотрены на уроке 10. Процедуры, так же как и макроМанды, цепны тем, что могут быть активизированы в любом месте программы. Процедурам, так же как и макроМандам, могут быть переданы некоторые аргументы, что позволяет, имея одну копию кода в памяти, изменять ее для каждого конкретного случая использования, хотя по гибкости использования процедуры уступают макроМандам. Особых проблем с разработкой и применением процедур нет. На уроке 10 нами были рассмотрены возможные варианты размещения процедур в программе:

- в начале программы (до первой исполняемой команды);
- в конце (после команды, возвращающей управление операционной системе);
- промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы. В этом случае необходимо предусмотреть обход процедуры с помощью команды безусловного перехода `jmp`;
- в другом модуле.

Главное условие здесь то, чтобы на процедуру не попадало управление. Три первых варианта из этих четырех относятся к случаю, когда процедуры находятся в одном сегменте кода. Мы их достаточно подробно обсудили. Что же касается последнего варианта, то он предполагает, что процедуры находятся в разных модулях. А это дает нам возможность говорить уже не об одной программе, а о нескольких. Эти программы должны быть связаны между собой по управлению и по данным. Если мы разберемся с тем, как организовать такую связь, то фактически сможем выполнить функциональную декомпозицию любой большой программы на несколько более мелких. В первой части данного урока мы рассмотрим, как организуется связь по управлению и по данным между программами на ассемблере. Во второй мы разберем связь между программами на ассемблере, Pascal и C.

Сначала необходимо отметить один общий для всех этих трех языков момент. Так как отдельный модуль в соответствии с концепцией модульного программирования — это функционально автономный объект, то он ничего не должен знать о внутреннем устройстве других модулей, и наоборот, другим модулям также ничего не известно о внутреннем устройстве данного модуля. Но должны быть какие-то средства, с помощью которых можно связать модули. В качестве аналогии можно привести известную вам организацию связи телевизора и видеомагнитофона через разъем типа «скарт». Связь унифицирована, то есть известно, что один контакт предназначен для видеосигнала, другой — для передачи звука и т. д. Телевизор и видеомагнитофон могут быть разными, но связь между ними одинакова. Та же идея лежит и в организации связи модулей. Внутреннее устройство модулей может совершенствоваться, они вообще могут в следующих версиях писаться на другом языке, но в процессе их объединения в единый исполняемый модуль этих особенностей не должно быть заметно. Таким образом, каждый модуль должен иметь такие средства, с помощью которых он извещал бы транслятор о том, что некоторый объект (процедура, переменная) должен быть видимым вне этого модуля. И наоборот, нужно объяснить транслятору, что некоторый объект находится вне данного модуля. Это позволит транслятору правильно сформировать машинные команды, оставив некоторые из поля незаполненными. Позднее, на этапе компоновки, программа TLINK или программа компоновки языка высокого уровня произведут настройку модулей и разрешат все внешние ссылки в объединяемых модулях.

Для того чтобы объявить о подобного рода видимых извне объектах, программа должна использовать две директивы TASM: `extrn` и `public`. Директива `extrn` предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено в директиве `public`. Директива `public` предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях. Синтаксис этих директив следующий:

`extrn ИМЯ:ТИП,..., ИМЯ:ТИП`

`public ИМЯ,...,ИМЯ`

Здесь имя — идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

- имена переменных, определенных директивами типа `db`, `dw` и т. д.;
- имена процедур;
- имена констант, определенных операторами `=` и `equ`.

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

- если имя — это имя переменной, то тип может принимать значения `byte`, `word`, `dword`, `pword`, `fword`, `qword` и `tbyte`;
- если имя — это имя процедуры, то тип может принимать значения `near` или `far`;

О если имя — это имя константы, то тип должен быть abs.

Покажем принцип использования директив extrn и public на схеме связи двух модулей Модуль 1 и Модуль 2 (см. листинги 14.1 и 14.2).

Листинг 14.1. Модуль 1

```
;Модуль 1
masm
.model small
.stack 256
.data
...
.code
my_proc_1 proc
...
my_proc_1 endp
my_proc_2 proc
...
my_proc_2 endp
;объявляем процедуру my_proc_1 видимой извне
public my_proc_1
start:
    mov ax,@data
...
endstart
```

Листинг 14.2. Модуль 2

```
;Модуль 2
masm
.model small
.stack 256
.data
...
.code
extrn my_proc_1      ;объявляем процедуру my_proc_1 внешней
start:
    mov ax,@data
...
    call my_proc_1      ;вызов my_proc_1 из модуля 1
endstart
```

Рассмотренная нами схема связи — это, фактически, связь по управлению. Но не менее важно организовать информационный обмен между модулями. Рассмотрим основные способы организации такой связи.

Организация интерфейса с процедурой

Прежде всего определимся с такими понятиями, как *аргумент*, *переменная*, *константа*. Они часто используются в контексте других понятий — модуля и процедуры, которые можно считать синонимами, хотя модуль — это все же более общее понятие.

Аргумент — это ссылка на некоторые данные, которые требуются для выполнения возложенных на модуль функций и размещенных вне этого модуля. По аналогии с макрокомандами рассматривают понятия *формального* и *фактического* аргументов. Исходя из этого, формальный аргумент можно рассматривать не как непосредственные данные или их адрес, а как «местодержатель» для действительных данных, которые будут подставлены в него с помощью фактического аргумента. Формальный аргумент можно рассматривать как элемент интерфейса модуля (конкретный вывод «скарта»), а фактический аргумент — это то, что фактически передается на место формального аргумента.

Переменная — это нечто, размещенное в регистре или ячейке памяти, что может в дальнейшем подвергаться изменению.

Константа — данные, значение которых никогда не изменяется.

Таким образом, если некоторые данные в модуле могут подвергаться изменению, то это *переменные*. Если переменная находится за пределами модуля (процедуры) и должна быть как-то передана в него, то для модуля она является формальным *аргументом*. Значение переменной передается в модуль для замещения соответствующего параметра при помощи фактического *аргумента*. Теперь, когда вы уяснили, чем отличаются формальные и фактические аргументы, мы далее будем называть их обобщенно — *аргументы*, а вы по контексту будете понимать, о каком виде аргументов идет речь.

Если входные данные для модуля — переменные, то один и тот же модуль можно использовать многократно для разных наборов значений этих переменных. Но как организовать передачу значений переменных в модуль (процедуру)? При программировании на языке высокого уровня программист ограничен в выборе способов передачи аргументов теми рамками, которые для него оставляет компилятор. В языке ассемблера практически нет никаких ограничений на этот счет, и фактически решение проблемы передачи аргументов предоставлено программисту.

Поэтому существуют следующие варианты передачи аргументов в модуль (процедуру):

- через регистры;
- через общую область памяти;
- через стек;
- с помощью директив `extern` и `public`.

Передача аргументов через регистры

Это наиболее простой в реализации способ передачи данных. Данные, переданные подобным способом, становятся доступными немедленно после передачи управления процедуре. Этот способ очень популярен при небольшом объеме передаваемых данных.

Ограничения на способ передачи аргументов через регистры:

- небольшое число доступных для пользователя регистров;
- нужно постоянно помнить о том, какая информация в каком регистре находится;

- ограничение размера передаваемых данных размерами регистра. Если размер данных превышает 8, 16 или 32 бита, то передачу данных посредством регистров произвести нельзя. В этом случае передавать нужно не сами данные, а указатели на них.

Такой способ передачи аргументов широко применяется при вызове функций DOS.

На уроке 13 (листинг 13.2) мы обсуждали программу, использующую макрокоманду, которая подсчитывала длину строки, оканчивающуюся символом \$. Для сравнения эффективности применения макрокоманд и процедур при программировании разработаем аналогичную программу (листинг 14.3), но с использованием процедуры подсчета количества символов в строке с конечным символом \$.

Для подсчета символов программа использует процедуру CountSymbol, расположенную в конце программы. Длина строки — не более 99 символов. Адрес строки передается процедуре как аргумент через регистр si. Результат подсчета возвращается в регистр bl и выводится на экран в вызывающей программе. Для вывода на экран используется прямой доступ к видеобуферу.

Листинг 14.3. Передача аргументов через регистры

```
;prog_14_3.asm
MASM
MODEL small ;модель памяти
STACK 256 ;размер стека
include iomac.inc ;подключение файла с макросами
.data ;начало сегмента данных
maskd db 71h ;маска вывода на экран
string db "Строка для подсчета $" ;тестовая строка
mesdb "В строке string"
cntdb 2 dup ("*") ;количество символов в строке
db " символов",10,13,'$'

.code
main proc ;точка входа в главную процедуру
    mov ax,@data
    mov ds,ax
;загрузка адреса строки
;(для передачи смещения в процедуру)
    lea si,string
;вызов процедуры
    call CountSymbol
    mov cl,bl ;счетчик для lodsb и stosw
    lea si,string ;в si – указатель на строку
    mov ax,0b800h
    mov es,ax ;загрузка в es адреса видеопамяти
    mov ah,maskd ;маска вывода на экран
    mov di,160 ;позиция вывода на экран
    cld ;просмотр вперед – для lodsb и stosw
disp:
    lodsb ;пересылка байта из ds:si в al
    stosw ;копирование значения ax
          ;в es:di (видеобуфер)
loop disp ;повтор цикла cx раз
```

```

;а теперь выведем количество символов в строке
mov al,b1
aam           ;в al две упакованные BCD-цифры
               ;результат подсчета
or   ax,3030h ;преобразование результата в код ASCII
mov cnt,ah
mov cnt+1,al
_OutStr    mes ;вывод строки mes
exit:
_exit          ;макрос выхода
main endp      ;конец главной процедуры

CountSymbol proc near
;процедура CountSymbol – подсчет символов в строке.
;На входе: si – смещение строки
;На выходе: b1 – длина в виде упакованного BCD-числа
push ax         ;сохранение используемых регистров
push cx
cld             ;просмотр вперед
mov cx,100      ;максимальная длина строки
;блок подсчета символов
go:
lodsb          ;загрузка символа строки в al
cmp al,'$'
je endstr
jcxz no_end
inc b1          ;приращение счетчика в b1 – количество
               ;подсчитанных символов в строке
loop go         ;повтор цикла
endstr:
pop cx          ;восстановление регистров из стека
pop ax
ret             ;возврат из процедуры
no_end:
;какие-то действия по обработке ситуации
;отсутствия в строке символа $
ret             ;возврат из процедуры
CountSymbol endp ;конец процедуры
end main       ;конец программы

```

Передача аргументов через общую область памяти

Этот вариант передачи аргументов предполагает, что вызывающая и вызываемая программы условились использовать некоторую область памяти как общую. Транслятор предоставляет специальное средство для организации такой области памяти. На уроке 5 мы разбирали директивы сегментации и их атрибуты. Один из них — атрибут комбинирования сегментов. Наличие этого атрибута указывает компоновщику TLINK, как нужно комбинировать сегменты, имеющие одно имя. Значение common означает, что все сегменты, имеющие одинаковое имя в объ-

единяемых модулях, будут располагаться компоновщиком, начиная с одного адреса оперативной памяти. Это значит, что они будут просто перекрываться в памяти и, следовательно, совместно использовать выделенную память.

Недостатком этого способа в реальном режиме работы микропроцессора является отсутствие средств защиты данных от разрушения, так как нельзя проконтролировать соблюдение правил доступа к этим данным. В защищенном режиме ситуация выглядит лучше. Этот режим мы рассмотрим на уроке 16.

Рассмотрим листинг 14.4 с примером использования общей области памяти для обмена данными между модулями. На этот раз программа состоит уже из двух независимых модулей, находящихся в разных файлах, и поэтому они представляют собой отдельные единицы трансляции. Функционально эти модули реализуют несложную задачу, которая заключается в том, что вызываемые процедуры формируют строку символов и передают ее через общую область, а вызывающая их процедура `main` выводит строку на экран.

Листинг 14.4. Передача аргументов через общую область памяти (модуль 1)

```
;prg14_4.asm
include mac.inc ;подключение файла с макросами
stksegment stack
    db 256 dup (0)
stkends
common_data segment para common "data" ;начало общей
;области памяти
bufdb 15 DUP (" ")
temp dw 0 ;буфер для хранения строки
common_data ends
extrn PutChar:far,PutCharEnd:far
code segment ;начало сегмента кода
assume cs:code,es:common_data
main proc
    mov ax,common_data
    mov es,ax
;вызов внешних процедур
    call PutChar
    call PutCharEnd
    push es
    pop ds
    _OutStr buf
exit:
    _Exit ;стандартный выход
main endp ;конец главной процедуры
code ends
end main
```

Вызываемые процедуры находятся в другом модуле (листинг 14.5).

Обратите внимание, что совсем не обязательно, чтобы данные в сегментах `common` имели одинаковые имена. Главное, и за этим нужно следить с особой тщательностью, — структура общих сегментов. Она должна быть абсолютно идентична во всех модулях данной программы, использующих обмен данными через общую память.

Листинг 14.5. Передача аргументов через общую область памяти (модуль 2)

```
:prg14_5.asm
include mac.inc ;подключение файла с макросами
stksegment stack
    db 256 dup (0)
stkends

pdata segment para public "data"
mesdb "Общий сегмент",0ah,0dh,'$'
temp1 db ?
temp2 dd ?
temp3 dq ?
pdata ends

public PutChar,PutCharEnd
common_data segment para common "data" ;начало общей
                                         ;области памяти
buffer db 15 DUP (" ") ;буфер для формирования строки
tmpSI dw 0
common_data ends
code segment ;начало сегмента кода
assume cs:code,es:common_data,ds:pdata
PutChar proc far ;объявление процедуры
    cld
    mov si,0
    mov buffer[si],'P'
    inc si
    mov buffer[si],'а'
    inc si
    mov buffer[si],'б'
    inc si
    mov buffer[si],'о'
    inc si
    mov buffer[si],'т'
    inc si
    mov buffer[si],'а'
    inc si
    mov buffer[si],'е'
    inc si
    mov buffer[si],'т'
    inc si
    mov buffer[si],'!'
    inc si
    mov tmpSI,si
    ret ;возврат из процедуры
PutChar endp ;конец процедуры
PutCharEnd proc far
    mov si,tmpSI
    mov buffer[si],'$'
    ret
PutCharEnd endp
code ends
end
```

Так как в данном примере программа состоит уже из двух модулей, то у читателя наверняка возник вопрос, как собрать ее в один исполняемый модуль. Можно предложить следующую последовательность шагов:

1. Выполнить трансляцию модуля `prg14_4.asm` и получить объектный модуль `prg14_4.obj`.
2. Выполнить трансляцию модуля `prg14_5.asm` и получить объектный модуль `prg14_5.obj`.
3. Скомпоновать программу утилитой TLINK командной строкой вида:
`tlink /v prg14_4.obj + prg14_5.obj`

В итоге будет создан исполняемый модуль `prg14_4.exe`. Вы можете исследовать этот модуль, используя отладчик, но имейте в виду следующее. В окне MODULE вы увидите только исходный текст программы `prg14_4.asm`. Для того чтобы войти по команде `call` в вызываемую процедуру, необходимо нажимать клавишу F7. Обработка этой команды приведет к открытию второго окна, в котором будет выведен текст вызванной процедуры.

Передача аргументов через стек

Этот способ наиболее часто используется для передачи аргументов при вызове процедур. Суть его заключается в том, что вызывающая процедура самостоятельно заносит в стек передаваемые данные, после чего производит вызов вызываемой процедуры. На уроке 10 мы рассматривали процессы, происходящие при передаче управления процедуре и возврате из нее. При этом мы обсуждали содержимое стека до и после передачи управления процедуре (см. рис. 10.4 и 10.5). Как следует из этих рисунков, при передаче управления процедуре микропроцессор автоматически записывает в вершину стека два (для процедур типа `near`) или четыре (для процедур типа `far`) байта. Вы помните, что эти байты являются адресом возврата в вызывающую программу. Если перед передачей управления процедуре командой `call` в стек были записаны переданные процедуре данные или указатели на них, то они окажутся под адресом возврата.

При рассмотрении архитектуры микропроцессора мы выяснили, что стек обслуживается тремя регистрами: `ss`, `sp` и `bp`. Микропроцессор автоматически работает с регистрами `ss` и `sp` в предположении, что они всегда указывают на вершину стека. По этой причине их содержимое изменять не рекомендуется. Для осуществления произвольного доступа к данным в стеке архитектура микропроцессора имеет специальный регистр `ebp\bp` (Base Point – указатель базы). Так же как и для регистра `esp\sp`, использование `ebp\bp` автоматически предполагает работу с сегментом стека. Перед использованием этого регистра для доступа к данным стека его содержимое необходимо правильно инициализировать, что предполагает формирование в нем адреса, который бы указывал непосредственно на переданные данные. Для этого в начало процедуры рекомендуется включить дополнительный фрагмент кода. Он имеет свое название – *пролог* процедуры. Типичный фрагмент программы, содержащий вызов процедуры с передачей аргументов через стек, может выглядеть так:

```

<1>      masm
<2>      model small
<3>...
<4>      proc_1 proc near ;«близкая» процедура (near) с n аргументами
<5>      ;начало пролога
<6>          push bp
<7>          mov bp,sp
<8>      ;конец пролога
<9>          mov ax,[bp+4]    ;доступ к аргументу arg_n для near-процедуры
<10>         mov ax,[bp+6]    ;доступ к аргументу arg_{n-1}
<11>         ...
<12>         ;команды процедуры
<13>         ;подготовка к выходу из процедуры
<14>         ;начало эпилога
<15>         mov sp,bp        ;восстановление sp
<16>         pop bp          ;восстановление значения старого bp
<17>         ret             ;до входа в процедуру
<18>         ;возврат в вызывающую программу
<19>         ;конец эпилога
<20>         proc_1 endp
<21>
<22>         .code
<23>         main proc
<24>             mov ax,@data
<25>             mov ds,ax
<26>             ...
<27>             push arg_1           ;запись в стек 1-го аргумента
<28>             push arg_2           ;запись в стек 2-го аргумента
<29>             ...
<30>             push arg_n           ;запись в стек n-го аргумента
<31>             call proc_1           ;вызов процедуры proc_1
<32>             ;действия по очистке стека после возврата из процедуры
<33>             ...
<34>             m1:
<35>             _exit
<36>         main endp
<37>         end main

```

Код пролога состоит всего из двух команд. Первая команда `push bp` сохраняет содержимое `bp` в стеке с тем, чтобы исключить порчу находящегося в нем значения в вызываемой процедуре. Вторая команда пролога `mov bp,sp` настраивает `bp` на вершину стека. После этого мы можем не волноваться о том, что содержимое `sp` перестанет быть актуальным, и осуществлять прямой доступ к содержимому стека. Что мы и делаем. Для доступа к `arg_n` достаточно сместиться от содержимого `bp` на 4, для `arg_{n-1}` — на 6 и т. д. Но эти смещения подходят только для процедур типа `near`. Для `far`-процедур эти значения необходимо скорректировать еще на 2, так как при вызове процедуры дальнего типа в стек записывается полный адрес — содержимое `cs` и `ip`. Поэтому для доступа к `arg_n` в строке 9 команда будет выглядеть так: `mov ax,[bp+6]`, а для `arg_{n-1}`, соответственно, `mov ax,[bp+8]` и т. д.

Конец процедуры также должен быть оформлен особым образом и содержать действия, обеспечивающие корректный возврат из процедуры. Фрагмент кода,

выполняющего такие действия, имеет свое название — *эпилог* процедуры. Код эпилога должен восстановить контекст программы в точке вызова вызываемой процедуры из вызывающей программы. При этом, в частности, нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, передававшиеся в процедуру. Это можно сделать несколькими способами:

- используя последовательность из n команд `pop xx`. Лучше всего это делать в вызывающей программе сразу после возврата управления из процедуры;
- откорректировать регистр указателя стека `sp` на величину $2*n$, например, командой `add sp, NN`, где $NN=2*n$, и n — количество аргументов. Это также лучше делать после возврата управления вызывающей процедуре;
- используя машинную команду `ret n` в качестве последней исполняемой команды в процедуре, где n — количество байт, на которое нужно увеличить содержимое регистра `esp\sp` после того, как со стека будут сняты составляющие адреса возврата. Видно, что этот способ аналогичен предыдущему, но выполняется автоматически микропроцессором.

В каком виде можно передавать аргументы в процедуру? Мы уже упоминали, что передаваться могут либо данные, либо их адреса (указатели на данные). В языке высокого уровня это называется передачей по *значению* и *адресу*, соответственно.

Наиболее простой способ передачи аргументов в процедуру — передача по значению. Этот способ предполагает, что передаются сами данные, то есть их значения. Вызываемая программа получает значение аргумента через регистр или через стек. Естественно, что при передаче переменных через регистр или стек на их размерность накладываются ограничения, связанные с размерностью используемых регистров или стека. Другой важный момент заключается в том, что при передаче аргументов по значению в вызываемой процедуре обрабатываются их копии. Поэтому значения переменных в вызывающей процедуре не изменяются.

Способ передачи аргументов по адресу предполагает, что вызываемая процедура получает не сами данные, а их адреса. В процедуре нужно извлечь эти адреса тем же методом, как это делалось для данных, и загрузить их в соответствующие регистры. После этого, используя адреса в регистрах, следует выполнить необходимые операции над самими данными. В отличие от способа передачи данных по значению, при передаче данных по адресу в вызываемой процедуре обрабатывается не копия, а оригинал передаваемых данных. Поэтому при изменении данных в вызываемой процедуре они автоматически изменяются и в вызывающей программе, так как изменения касаются одной области памяти.

Использование директив `extern` и `public`

Эти директивы мы уже упоминали выше, когда рассматривали варианты взаимного расположения вызывающей программы и вызываемой процедуры. Ко всему сказанному добавим, что директивы `extern` и `public` также можно использовать для обмена информацией между модулями. Назначение и форматы этих директив уже были рассмотрены, поэтому сейчас опишем только порядок их использования для обмена данными. Выделим несколько вариантов их применения:

1. Оба модуля используют сегмент данных вызывающей программы.
2. У каждого из модулей есть свой собственный сегмент данных.
3. Модули используют атрибут комбинирования (объединения) сегментов `private` в директиве сегментации `segment`.

Рассмотрим эти варианты на примере программы, которая определяет в сегменте данных две символьные переменные и вызывает процедуру, выводящую эти символы на экран.

Вариант 1. Два модуля используют только сегмент данных вызывающей программы (листинги 14.6 и 14.7).

В этом случае не требуется переопределения сегмента данных в вызываемой процедуре. В листинге 14.6 в вызывающей программе определены две переменные, вывод на экран которых осуществляется вызываемой программой (листинг 14.7).

Листинг 14.6. Вариант 1 использования директив `extrn` и `public` (Модуль 1)

```
:prg14_6.asm
;Вызывающий модуль
include mac.inc
extrn my_proc2:far
public per1,per2
stk segment stack
    db 256 dup (0)
stk ends
data segment
per1 db "1"
per2 db "2"
data ends
code segment
main proc far
assume cs:code,ds:data,ss:stk
    mov ax,data
    mov ds,ax
    call my_proc2
    exit
main endp
code ends
end main
```

Листинг 14.7. Вариант 1 использования директив `extrn` и `public` (Модуль 2)

```
:prg14_7.asm
;Вызываемый модуль
include mac.inc
extrn per1:byte,per2:byte
public my_proc2
code segment
my_proc2 proc far
assume cs:code
;вывод символов на экран
    mov dl,per1
    OutChar
    mov dl,per2
```

```
OutChar  
ret  
my_proc2 endp  
code    ends  
end
```

Сборка программы из двух модулей для этого и следующих вариантов осуществляется аналогично листингам 14.4 и 14.5.

Вариант 2. У каждого из модулей есть свой собственный сегмент данных.

В этом случае для доступа к разделяемым переменным из другого модуля требуется переопределение сегмента данных в вызываемой процедуре (строки 17–19 и 23–24 листинга 14.8).

Листинг 14.8. Вариант 2 использования директив extrn и public

```
:prgl4_8.asm  
;Вызывающий модуль – тот же, что и для предыдущего варианта.  
<1>;Вызываемый модуль  
<2>include      mac.inc  
<3>extrn per1:byte,per2:byte  
<4>public my_proc2  
<5>data   segment  
<6>per0  db    "0"  
<7>data   ends  
<8>code   segment  
<9>my_proc2  proc far  
<10>    assume cs:code,ds:data  
<11>    ;вывод символов на экран  
<12>        mov    ax,data  
<13>        mov    ds,ax  
<14>        mov    dl,per0  
<15>        OutChar  
<16>        push   ds          ;сохранили ds  
<17>        mov    ax,seg per1 ;сегментный адрес per1 в ds  
<18>        mov    ds,ax  
<19>        mov    dl,per1  
<20>        OutChar          ;вывод per1  
<21>        mov    dl,per2  
<22>        OutChar          ;вывод per2  
<23>        pop    ds          ;восстановили ds  
<24>        mov    dl,per0  
<25>        OutChar          ;и еще раз per0  
<26>        ret  
<27> my_proc2 endp  
<28> code    ends  
<29> end
```

Вариант 2а. У каждого из модулей есть свой собственный сегмент данных (листинг 14.9).

Это несколько улучшенный вариант предыдущего примера, где мы использовали для адресации данных в разных сегментах данных один регистр ds. В этом случае для доступа к разделяемым переменным из другого модуля используется

один из дополнительных сегментных регистров данных, к примеру es. Заметьте, что обращение к данным другого сегмента осуществляется с использованием префикса замены сегмента (строки 18 и 20).

Листинг 14.9. Вариант 2а использования директив extrn и public

```
:prg14_9.asm
;Вызывающий модуль – тот же, что и для предыдущего варианта.
<1>      ;Вызываемый модуль
<2>      include    iomac.inc
<3>      extrn per1:byte.per2:byte
<4>      public my_proc2
<5>      data segment
<6>      per0 db "0"
<7>      data ends
<8>      code segment
<9>      my_proc2 proc far
<10>     assume cs:code,ds:data
<11>     ;вывод символов на экран
<12>     mov ax,data
<13>     mov ds,ax
<14>     mov dl,per0
<15>     _OutChar
<16>     mov ax,seg per1
<17>     mov es,ax
<18>     mov dl,es:per1
<19>     _OutChar
<20>     mov dl,es:per2
<21>     _OutChar
<22>     mov dl,per0
<23>     _OutChar
<24>     ret
<25>     my_proc2 endp
<26>     code ends
<27>     end
```

Вариант 3. Использование атрибута комбинирования (объединения) сегментов public в директиве сегментации segment для сегментов данных модулей (листинги 14.10 и 14.11).

Данное значение атрибута комбинирования заставляет компоновщик объединить последовательно сегменты с одинаковыми именами. Все адреса и смещения будут вычисляться относительно начала этого нового сегмента. В этом случае не понадобится производить дополнительной настройки сегментных регистров (как было в двух предыдущих случаях).

Листинг 14.10. Вариант 3 использования директив extrn и public (Модуль 1)

```
:prg14_10.asm
;Вызывающий модуль
<1> include    mac.inc
<2> extrn my_proc2:far,per0:byte
<3> public per1,per2
<4> stk segment stack
```

```

<5>    db    256 dup (0)
<6>stk ends
<7>data segment para public "data"
<8>per1 db    "1"
<9>per2 db    "2"
<10>   data ends
<11>   code segment
<12>   main proc far
<13>   assume cs:code,ds:data,ss:stk
<14>       mov    ax,data
<15>       mov    ds,ax
<16>       mov    d1,per0
<17>       OutChar
<18>       call   my_proc2
<19>       exit
<20>   main endp
<21>   code ends
<22>   end   main

```

Листинг 14.11. Вариант 3 использования директив extrn и public (Модуль 2)

```

<1>;prg14_11.asm
<2>;Вызываемый модуль
<3>include mac.inc
<4>extrn per1:byte,per2:byte
<5>public my_proc2,per0
<6>data segment para public "data"
<7>per0 db "0"
<8>data ends
<9>code segment
<10>  my_proc2 proc far
<11>  assume cs:code,ds:data
<12>  ;ds загружать не надо, так как компоновщик его присоединит
<13>  ;к сегменту данных первого модуля
<14>  ;вывод символов на экран
<15>      mov   d1,per0
<16>      OutChar
<17>      mov   d1,per1
<18>      OutChar
<19>      mov   d1,per2
<20>      OutChar
<21>      mov   d1,per0
<22>      OutChar
<23>      ret
<24>  my_proc2 endp
<25>  code ends
<26>  end

```

Возврат результата из процедуры

В отличие от языков высокого уровня, в языке ассемблера нет отдельных понятий для процедуры и функции. Организация возврата результата из процедуры полностью ложится на программиста. Как это сделать, для внимательного чита-

теля, наверное, уже понятно, ибо получение результата тоже можно рассматривать как передачу аргументов. Поэтому мы ограничимся кратким пояснением. В общем случае программист располагает тремя вариантами возврата значений из процедур:

- С использованием регистров. Ограничения здесь те же, что и при передаче данных, — это небольшое количество доступных регистров и их фиксированный размер. Функции DOS используют именно этот способ. Из рассматриваемых здесь трех вариантов данный способ является наиболее быстрым, поэтому его есть смысл использовать для организации критичных по времени вызова процедур.
- С использованием общей области памяти. Этот способ удобен при возврате большого количества данных, но требует внимательности в определении областей данных и подробного документирования для устранения неоднозначностей.
- С использованием стека. Здесь, подобно передаче аргументов через стек, также нужно использовать регистр `bp`. При этом возможны следующие варианты:
 - использование для возвращаемых аргументов тех же ячеек в стеке, которые применялись для передачи аргументов в процедуру. То есть предполагается замещение ставших ненужными входных аргументов выходными данными;
 - предварительное помещение в стек наряду с передаваемыми аргументами фиктивных аргументов с целью резервирования места для возвращаемого значения. При использовании этого варианта процедура, конечно же, не должна пытаться очистить стек командой `ret`. Эту операцию придется делать в вызывающей программе, например командой `pop`.

В ходе вышеприведенного обсуждения мы выяснили, что язык ассемблера не накладывает никаких ограничений на организацию процесса передачи данных и возврата значений между двумя процедурами, а в более общем случае — и между модулями, представляющими отдельные файлы. Наиболее быстрый способ такого обмена — использование регистров. Но часто требуется связывать между собой не только программы, написанные на ассемблере, но и программы на разных языках. В этом случае универсальным становится способ обмена данными с использованием стека. Ниже мы рассмотрим, как решается проблема связи программных модулей, написанных на языке ассемблера и языках высокого уровня.

Связь ассемблера с языками высокого уровня

На протяжении всей книги мы неоднократно подчеркивали сильные и слабые стороны языка ассемблера как языка программирования. Писать на нем достаточно объемные программы утомительно. И всегда ли это нужно? Конечно, не всегда. Если программа не предназначена для решения каких-то системных задач, требующих максимально эффективного использования ресурсов компьютера, если

к ней не предъявляются сверхжесткие требования по размеру и времени работы, если вы не «фанат» ассемблера — то, на мой взгляд, следует подумать о выборе одного из языков высокого уровня. Существует и третий, компромиссный путь — комбинирование программ на языке высокого уровня с кодом на ассемблере. Такой способ обычно используют в том случае, если в вашей программе есть участки, которые либо невозможно реализовать без использования ассемблера, либо его применение может значительно повысить эффективность работы программы. Большинство компиляторов учитывают возможность комбинирования их «родного» кода с ассемблером. Как именно? Это зависит от конкретного компилятора языка высокого уровня, поэтому нет смысла рассматривать данный вопрос абстрактно. Учитывая, что большинство программистов работают или, по крайней мере, владеют основами программирования для языков C и Pascal, дальнейшее обсуждение будет вестись для компиляторов этих языков: Borland Pascal 7.0 и Visual C++ 4.0. Но не стоит думать, что если вы используете компиляторы других фирм или других версий, то информация, приведенная ниже, вам не понадобится. Принципиальные моменты останутся теми же. Единственное, что вам нужно будет сделать, — это уточнить в документации некоторые опции и, возможно, особенности организации связи с кодом на ассемблере. Так что вы можете быть уверены в актуальности информации, представленной ниже. Она далеко не избыточна. Ее назначение — лишь отразить наиболее принципиальные моменты связи программ на языках Pascal и C с ассемблером.

Вначале мы отметим общие моменты, актуальные как для языка C, так и для Pascal. Затем на примерах конкретных программ мы обсудим моменты, специфичные для каждого из этих языков.

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

- Использование операторов типа `inline`¹ и ассемблерных вставок. Эта форма сильно зависит от синтаксиса языка высокого уровня и конкретного компилятора. Она предполагает, что ассемблерные коды в виде команд ассемблера или прямо в машинных командах вставляются в текст программы на языке высокого уровня. Компилятор языка распознает их как команды ассемблера (машинные коды) и без изменений включает в формируемый им объектный код. Эта форма удобна, если надо вставить небольшой фрагмент.
- Использование внешних процедур и функций. Это более универсальная форма комбинирования. У нее есть ряд преимуществ:
 - написание и отладку программ можно производить независимо;
 - написанные подпрограммы можно использовать в других проектах;
 - облегчаются модификация и сопровождение подпрограмм в течение жизненного цикла проекта. К примеру, если ваша процедура на ассемблере производит работу с некоторым внешним устройством, то при смене уст-

¹ Под операторами типа `inline` здесь понимаются средства языка высокого уровня, подобные оператору `inline()` языка Pascal. В скобках указывается строка машинных кодов. Для получения такой строки целесообразно написать нужный фрагмент на ассемблере, скомпилировать исполняемый модуль, а затем запустить отладчик. В окне CPU отладчика вы увидите машинные коды ваших инструкций; их нужно переписать и вставить в скобки оператора `inline`. При этом нужно учитывать синтаксис языка.

ройства вам нет необходимости перекраивать весь проект — достаточно заменить только работающую с ним процедуру, оставив неизменным интерфейс программы на языке высокого уровня с ассемблером.

Так как использование операторов `inline` и ассемблерных вставок сильно зависит от синтаксиса конкретного языка, то мы рассматривать их не будем, а уделим основное внимание связи через внешние процедуры и функции. Как вы понимаете, здесь возможны два вида связи — программа на языке высокого уровня вызывает процедуру на ассемблере и наоборот. Здесь мы также ограничимся рассмотрением связи только в одну сторону, когда программа на языке высокого уровня вызывает процедуру на ассемблере. Это наиболее часто используемый вид связи. Вспомним синтаксис директивы `rgos`, введенной на уроке 10:

`имя_процедуры PROC [[модификатор_языка]язык] [расстояние]`

Один из операндов — `язык`. Он служит для того, чтобы компилятор мог правильно организовать интерфейс (связь данных) между процедурой на ассемблере и программой на языке высокого уровня. Необходимость такого указания возникает вследствие того, что способы передачи аргументов при вызове процедур различны для разных языков высокого уровня.

TASM поддерживает несколько значений операнда `язык`. В табл. 14.1 для некоторых из этих значений приведены характерные особенности передачи аргументов и соглашения о том, какая процедура очищает стек — вызывающая или вызываемая. Под *направлением передачи аргументов* понимается порядок, в котором аргументы включаются в стек, по сравнению с порядком их следования в вызове процедуры. Так, к примеру, для языка Pascal характерен прямой порядок включения аргументов в стек: первым в стек записывается первый передаваемый аргумент из оператора вызова процедуры, вторым — второй аргумент и т. д. На вершине стека после записи всех передаваемых аргументов оказывается последний аргумент. Для языка C, наоборот, характерен обратный порядок передачи аргументов. В соответствии с ним в стек сначала включается последний аргумент из оператора вызова процедуры (или функции), затем предпоследний и т. д. В конечном итоге на вершине стека оказывается первый аргумент. Что же касается очистки стека, то понятно, что должны быть определенные договоренности об этом. В языке Pascal эту операцию всегда совершают вызываемая процедура, в языке C — вызывающая. При разработке программы с использованием только одного языка высокого уровня об этом задумываться не имеет смысла, но если мы собираемся связывать несколько разноязыковых модулей, то эти соглашения нужно иметь в виду.

Вспомните действия, которые мы делали для того, чтобы настроиться на аргументы в стеке. Теперь, указывая язык, с программой на котором будет осуществляться связь, все действия по настройке стека будут производиться компилятором. При этом в текст процедуры он включит дополнительные команды входа в процедуру (пролог) и выхода из нее (эпилог). При этом код эпилога может повторяться несколько раз — перед каждой командой `ret`. Для значения `NOLANGUAGE` и по умолчанию коды пролога и эпилога не создаются.

Для пояснения последних замечаний рассмотрим на конкретных примерах организацию связей между модулями на ассемблере и модулями наиболее популяр-

ных языков высокого уровня — С и Pascal. Просьба к читателю — не воспринимать нижеследующие примеры как образец оптимальности, а рассматривать их лишь как учебные для иллюстрации принципов организации межъязыковых связей.

Таблица 14.1. Передача аргументов в языках высокого уровня

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

Связь Pascal—ассемблер

Организацию такой связи рассмотрим на примере следующей задачи: разработаем программу на языке Pascal, которая выводит символ заданное количество раз, начиная с определенной позиции на экране (листинги 14.12 и 14.13). Все числовые аргументы определяются в программе на Pascal. Вывод символа осуществляется процедурой ассемблера. Очевидно, что основная проблема в этой задаче — организация взаимодействия модулей на Pascal и ассемблере.

Листинг 14.12. Взаимодействие Pascal—ассемблер (модуль на Pascal)

```
<1> {prg14_12.pas}
<2> {Программа, вызывающая процедуру на ассемблере}
<3> program my_pas;
<4> {$D+} {включение полной информации для отладчика}
<5> uses crt;
<6> procedure asmproc(ch:char;x,y,kol:integer); external;
<7> {процедура asmproc объявлена как внешняя}
<8> {$L c:\bp\work\prg14_12.obj}
<9> BEGIN
<10>    clrscr;      {очистка экрана}
<11>    asmproc("a",1,4,5);
<12>    asmproc("s",9,2,7);
<13> END.
```

Листинг 14.13. Взаимодействие Pascal—ассемблер (модуль на ассемблере)

```
<1> :prg14_13.asm
<2> ;Процедура на ассемблере, которую вызывает
<3> ;программа на Pascal.
<4> ;Для вывода на экран используются службы BIOS:
<5> ;02h – позиционирование курсора.
<6> ;09h – вывод символа заданное количество раз.
<7> MASM
<8> MODEL small
<9> STACK 256
<10> .code
<11> asmproc proc near
<12> PUBLIC asmproc ;объявлена как внешняя
<13>     push bp ;пролог
<14>     mov bp,sp
<15>     mov dh,[bp+6] ;номер строки для вывода
<16>                 ;символа у – в dh
<17>     mov dl,[bp+8] ;номер столбца для вывода
<18>                 ;символа х – в dl
<19>     mov ah,02h ;номер службы BIOS
<20>     int 10h ;вызов прерывания BIOS
<21> ;вызов функции 09h прерывания BIOS 10h:
<22> ;вывод символа из a1 на экран
<23>     mov ah,09h ;номер службы BIOS
<24>     mov a1,[bp+10] ;символ ch в a1
<25>     mov b1,07h ;атрибут символа – в b1
<26>     xor bh,bh
<27>     mov cx,[bp+4] ;количество «выводов»
<28>                 ;символа – в cx
<29>     int 10h ;вызов прерывания BIOS
<30>     pop bp ;восстановление bp
<31> ;очистка стека и возврат из процедуры
<32>     ret 8
<33> asmproc endp ;конец процедуры
<34> end
```

Типовая схема организации такой связи следующая:

- Написать процедуру на ассемблере дальнего (*far*) или ближнего типа (*near*). Назовем ее для примера *asmproc*. В программе на языке ассемблера (назовем ее *prg14_13.asm*), в которую входит процедура *asmproc*, необходимо объявить имя этой процедуры внешним с помощью директивы *PUBLIC*:

```
PUBLIC asmproc
```

Для того чтобы процедура на ассемблере при компоновке с программой на Pascal воспринималась компилятором Borland Pascal 7.0 как *far* или *near*, недостаточно будет просто объявить ее таковой в директиве *proc* (строка 11 листинга 14.13). Кроме того, вам нужно включить или выключить опцию компилятора, доступную через меню интегрированной среды: *Options > Compiler > Force far calls*. Установка данной опции заставляет компилятор генерировать дальние вызовы подпрограмм. Альтернатива данной опции – ключ в программе *{\$F+}* или *{\$F-}* (соответственно,

включено или выключено). Это — локальные ключи, то есть в исходном тексте программы на Pascal их может быть несколько, и они могут, чередуясь друг с другом, поочередно менять форму генерируемых адресов перехода — для одних подпрограмм — дальние вызовы, для других — ближние.

- Произвести компиляцию программы `prg14_13.asm` с целью устранения синтаксических ошибок и получения объектного модуля программы:

```
prg14_13.obj:  
tasm /zi prg14_13...
```

- В программе на Pascal `prg14_12.pas`, которая будет вызывать внешнюю процедуру на ассемблере, следует вставить директиву компилятора `{$L \путь\prg_14_12.obj}`. Эта директива заставит компилятор в процессе компиляции программы `prg14_12.pas` загрузить с диска объектный модуль программы `prg14_12.obj`. В программе `prg14_12.pas` необходимо объявить процедуру `asmproc` как внешнюю. В итоге последние два объявления в программе на Pascal будут выглядеть так:

```
{$L my_asm}  
procedure asmproc(ch:char;kol,x,y:integer); external;
```

- если вы собираетесь исследовать в отладчике работу программы, то необходимо потребовать, чтобы компилятор включил отладочную информацию в генерируемый им исполняемый модуль. Для этого есть две возможности. Первая заключается в использовании глобального ключа `{$D+}`. Этот ключ должен быть установлен сразу после заголовка программы на Pascal. Вторая, альтернативная возможность заключается в установке опции компилятора: **Options ▶ Compiler ▶ Debug Information**;

- выполнить компиляцию программы на Pascal. Для компиляции удобно использовать интегрированную среду. Для изучения особенностей связки Pascal — ассемблер удобно прямо в интегрированной среде перейти к работе в отладчике через меню **Tools ▶ Turbo Debugger** (или клавишами Shift+F4). Будет загружен отладчик. Его среда вам хорошо знакома; в данном случае в окне **Module** вы увидите текст программы на Pascal. Нажимая клавишу F7, вы в пошаговом режиме будете исполнять программу на Pascal. Когда очередь дойдет до вызова процедуры на ассемблере, отладчик откроет окно с текстом программы на ассемблере. Но наш совет вам — не ждать этого момента, так как вы уже пропустили некоторые интересные вещи. Дело в том, что отладчик скрывает от вас момент перехода из программы на Pascal в процедуру на ассемблере. Поэтому лучше всего исполнять программу при открытом окне CPU отладчика. И тогда вы станете свидетелями тех процессов, которые мы будем обсуждать ниже.

Если бы взаимодействие программ ограничивалось только передачей и возвратом управления, то на этом обсуждение можно было бы и закончить. Но дело значительно усложняется, когда требуется передать аргументы (в случае процедуры) или передать аргументы и возвратить результат (в случае функции). Рассмотрим процессы, которые при этом происходят.

Передача аргументов при связи модулей на разных языках всегда производится через стек. Компилятор Pascal генерирует соответствующие команды при обра-

ботке вызова процедуры ассемблера. Это как раз те команды, которые отладчик пытается скрыть от нас. Они записывают в стек аргументы и генерируют команду `call` для вызова процедуры ассемблера. Чтобы убедиться в этом, просмотрите исполнительный код программы в окне CPU отладчика. После обработки вызова процедуры и в момент передачи управления процедуре `asmproc` содержимое стека будет таким, как показано на рис. 14.1, *а*. Для доступа к этим аргументам можно использовать различные методы; наиболее удобный из них — использование регистра `bp`.

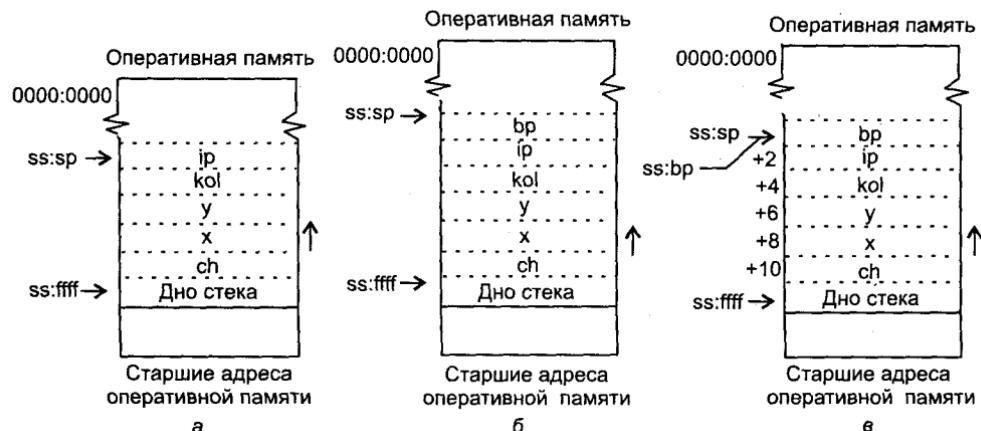


Рис. 14.1. Изменение содержимого стека при передаче управления Pascal—ассемблер

Регистр `bp`, как уже отмечалось, специально предназначен для организации произвольного доступа к стеку. Когда мы рассматривали связь ассемблерных модулей, то говорили о необходимости добавления в текст вызываемого модуля фрагментов, настраивающих его на передаваемые ему аргументы. При организации связи разноязыковых модулей также нужно вставлять подобные дополнительные фрагменты кода. Они, кроме всего прочего, позволяют учесть особенности конкретного языка. Фрагмент, вставляемый в самое начало вызываемого модуля, называется *прологом* модуля (процедуры). Фрагмент, вставляемый перед командами передачи управления вызывающему модулю, называется *эпилогом* модуля (процедуры). Его назначение — восстановление состояния вычислительной среды на момент вызова данного модуля.

Рассмотрим действия, выполняемые кодами пролога и эпилога при организации связи Pascal—ассемблер.

Действия, выполняемые кодом пролога:

- Сохранить значение `bp` в стеке. Это делается с целью запоминания контекста вызывающего модуля. Стек при этом будет выглядеть, как на рис. 14.1, *б*.
- Записать содержимое `sp` в `bp`. Тем самым `bp` теперь тоже будет указывать на вершину стека (рис. 14.1, *в*).

После написания кода пролога все обращения к аргументам в стеке можно организовывать относительно содержимого регистра `bp`. Из рис. 14.1, *в* видно, что для

обращения к верхнему и последующим аргументам в стеке содержимое bp необходимо откорректировать. Нетрудно посчитать, что величина корректировки будет отличаться для процедур дальнего (far) и ближнего (near) типов. Причина понятна: при вызове near-процедуры, в зависимости от установленного режима адресации — use16 или use32, — в стек записывается 2/4 байта в качестве адреса возврата (содержимое ip/eip), а при вызове far-процедуры в стек записывается 4/8 байта (содержимое ip/eip и cs)¹.

Таким образом, коды пролога для near- и far-процедур, соответственно, будут выглядеть следующим образом:

```
asmproc proc near  
;пролог для near-процедуры  
    push bp  
    mov bp,sp  
;к прологу можно добавить команду  
;корректировки bp на 4, с тем, чтобы регистр bp  
;указывал на верхний из передаваемых аргументов в стеке  
    add bp,4 ;теперь bp указывает на k01  
  
...  
asmproc proc far  
;пролог для far-процедуры  
    push bp  
    mov bp,sp  
;к прологу можно добавить команду  
;корректировки bp на 6, с тем, чтобы регистр bp  
;указывал на верхний из передаваемых аргументов в стеке  
    add bp,6 ;теперь bp указывает на k01  
...
```

Далее доступ к переданным в стеке данным осуществляется, как показано в листинге 14.7.

Как видите, все достаточно просто. Но если мы вдруг решили изменить тип нашей процедуры ассемблера с far на near или наоборот, то нужно явно изменить и код пролога. Это не совсем удобно. TASM предоставляет выход в виде директивы ARG, которая служит для работы с аргументами процедуры. Формат директивы ARG следующий (рис. 14.2).

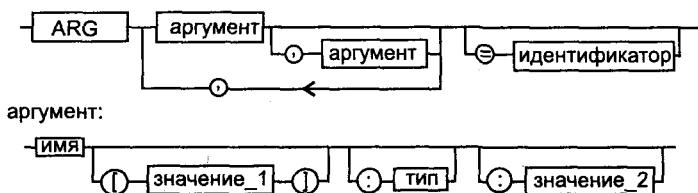


Рис. 14.2. Синтаксис директивы ARG

¹ Если действует режим адресации use32, то в стек записываются двойные слова. По этой причине запись 16-разрядного регистра cs также производится четырьмя байтами, при этом два старших байта этого значения — нулевые.

Несколько слов об обозначениях на рис. 14.2:

имя — идентификатор переменной, который будет использоваться в процедуре на ассемблере для доступа к соответствующей переменной в стеке;

тип — определяет тип данных аргумента (по умолчанию `word` для `use16` и `dword` для `use32`);

значение_1 — определяет количество аргументов с данным именем. Место в стеке для них будет определено, исходя из расчета: **значение_1 * значение_2 * (размер_типа)**. По умолчанию **значение_1 = 1**;

значение_2 — определяет, сколько элементов данного типа задает данный аргумент. По умолчанию его значение равно 1, но для типа `byte` **значение_2 = 2**, так как стековые команды не работают с отдельными байтами. Хотя, если явно задать **значение_2 = 1**, то транслятор действительно будет считать, что в ячейку стека помещен один байт;

идентификатор — имя константы, значение которой присваивает транслятор. Об идентификаторе мы подробно поговорим чуть ниже.

Таким образом, директива `ARG` определяет аргументы, передаваемые в процедуру. Ее применение позволяет обращаться к аргументам по их именам, а не смещениям относительно содержимого `bp`. К примеру, если в начале рассматриваемой нами процедуры на ассемблере `asmproc` задать директиву `ARG` в виде `arg kol:word,y:word,x:word,chr:byte`, то к аргументам процедуры можно будет обращаться по их именам, без подсчета смещений. Ассемблер сам выполнит всю необходимую работу. В этом можно убедиться, запустив программу в отладчике. Обратите внимание: порядок следования аргументов в директиве `arg` является обратным порядку их следования в описании процедуры (строка `procedure asmproc(ch:char;x,y,kol:integer); external;` в программе на Pascal). Процедура `asmproc` с использованием директивы `arg` представлена в листинге 14.14.

После того как решена проблема передачи аргументов в процедуру и выполнены все необходимые действия, возникает очередной вопрос — как правильно возвратить управление? При возврате управления в программу на Pascal нужно помнить, что соглашения этого языка требуют, чтобы вызываемые процедуры самостоятельно очищали за собой стек. Программа на ассемблере также должна удовлетворять этому требованию и заботиться об очистке стека перед своим завершением. Для этого необходимо составить эпилог.

Действия, выполняемые кодом эпилога для связи Pascal—ассемблер:

1. Записать содержимое `bp` в `sp` командой `mov sp, bp`. Это действие восстанавливает в `sp` значение, которое было на момент входа в процедуру. Необходимость в этом действии возникает в том случае, если в процедуре производилась работа со стеком. В листинге 14.3 такой работы не было, поэтому код эпилога содержит только действия следующих пунктов.

2. Восстановить сохраненный в стеке регистр `bp`.

3. Удалить из стека переданные процедуре аргументы.

Для удаления из стека аргументов можно использовать различные способы:

О явно скорректировать значение `sp`, переместив указатель стека на необходимое количество байт в положительную сторону. Это — не универсальный

способ, к тому же он чреват ошибками, особенно при частых модификациях программы;

- использовать в директиве `arg` после записи последнего аргумента операнд, состоящий из символа равенства «=» и идентификатора, указанного за ним в следующей синтаксической конструкции: `=идентификатор`. В этом случае TASM при обработке директивы `arg` подсчитает количество байт, занятых всеми аргументами, и присвоит их значение `идентификатору`. В нашем случае директиву `arg` можно определить так: `arg ch:byte;x:word;y:word;kol:word=a_size`. TASM после обработки данной директивы присвоит имени `a_size` значение 8 (байт). Это имя впоследствии нужно будет указать в качестве операнда команды `ret`:

```
ret a_size
```

Листинг 14.14. Использование директивы `arg`

```
{prg14_14.pas}
{Программа на Pascal, вызывающая процедуру на ассемблере, полностью совпадает
с листингом 14.12}
:prg14_14.asm
MASM
MODEL    small
STACK    256
.code
main:
asmproc proc near
;объявление аргументов:
arg      kol:WORD,y:WORD,x:WORD,chr:BYTE=a_size
PUBLIC   asmproc
push    bp          ;сохранение указателя базы
mov     bp,sp       ;настройка bp на стек через sp
mov     dh,byte ptr y  ; y в dh
mov     dl,byte ptr x  ; x в dl
mov     ah,02h       ;номер службы BIOS
int    10h          ;вызов прерывания BIOS
mov     ah,09h       ;номер службы BIOS
mov     al,chr        ;символ – в al
mov     bl,07h       ;маска вывода символа
xor     bh,bh
mov     cx,kol       ;kol в cx
int    10h          ;вызов прерывания BIOS
pop     bp          ;эпилог
ret     a_size        ;будет ret 8 и выход из процедуры
asmproc endp         ;конец процедуры
end main            ;конец программы
```

Есть еще одна возможность организации данных Pascal—ассемблер — использование operandов директивы `model`. Вы помните, что она позволяет задать модель памяти и учесть соглашения языков высокого уровня о вызове процедур. Для связи Pascal — ассемблер ее можно задавать в виде:

```
MODEL large,pascal
```

Задание в таком виде директивы `model` позволяет:

- описать аргументы процедуры непосредственно в директиве `proc`:

```
asmproc proc near ch:byte,x:word,y:word,kol:word
```

- автоматически сгенерировать код пролога и эпилога в процедуре на ассемблере;
- для доступа к аргументам, объявленным в proc, использовать их имена. В этом отношении данный вариант является аналогом предыдущего варианта с директивой arg.

Листинг 14.15 демонстрирует, как отразились особенности данного варианта на тексте процедуры ассемблера. Обратите внимание, что пролог уже нет, так как он формируется транслятором автоматически; вместо эпилога обязательно нужно задавать только ret без операндов. Интересно изучить текст листинга 14.16, который получится в результате трансляции листинга 14.15. В нем видны сформированные транслятором коды пролога и эпилога. Кроме того, транслятор заменил команду ret без операндов командой ret 0008, которая, в соответствии с требованиями к взаимодействию с программами на Pascal, удалит из стека аргументы, переданные вызываемой процедуре.

Листинг 14.15. Использование директивы MODEL

```
{prg14_15.pas}
{Программа на Pascal, вызывающая процедуру на ассемблере, полностью совпадает
с листингом 14.12}
;prg14_15.asm
MASM
MODEL large,pascal
STACK 256
.code
main:
asmproc proc near chr:BYTE,x:WORD,y:WORD,kol:WORD
PUBLIC
asmproc
    mov dh,byte ptr y      ; у в dh
    mov dl,byte ptr x      ; x в dl
    mov ah,02h              ;номер службы BIOS
    int 10h                ;вызов прерывания BIOS
    mov ah,09h              ;номер службы BIOS
    mov al,chr               ;символ – в al
    mov bl,07h              ;маска вывода символа
    xor bh,bh
    mov cx,kol             ;kol в cx
    int 10h                ;вызов прерывания BIOS
    ret
asmproc endp                 ;конец процедуры
end main                     ;конец программы
```

Листинг 14.16. Результат трансляции листинга 14.15

Turbo Assembler Version 4.1 17/04/98 22:30:57 Page 1

prg14_82.asm

```
1                               ;prg14_92.asm
2
3 0000   MASM
4 0000   MODEL large,pascal
5 0000   STACK 256
6 0000   .code
7 0000   main:
         .asmproc proc near chr:BYTE,x:WORD,y:WORD,kol:WORD
```

```

8      PUBLIC asmproc
1 9      0000 55    PUSH   BP
1 10     0001 8B EC  MOV    BP,SP
1 11     0003 8A 76 06  mov    dh,byte ptr y      ; y в dh
1 12     0006 8A 56 08  mov    d1.byte ptr x      ; x в d1
1 13     0009 B4 02  mov    ah,02h                ;номер службы BIOS
1 14     000B CD 10  int    10h                  ;вызов прерывания BIOS
1 15     000D B4 09  mov    ah,09h                ;номер службы BIOS
1 16     000F 8A 46 0A  mov    a1.chr               ;символ – в a1
1 17     0012 B3 07  mov    b1,07h                ;маска вывода символа
1 18     0014 32 FF  xor    bh,bh
1 19     0016 8B 4E 04  mov    cx,kol ;kol в cx
1 20     0019 CD 10  int    10h                  ;вызов прерывания BIOS
1 21     001B 5D    POP    BP
1 22     001C C2 0008 RET   00008h
1 23     001F         asmproc  endp               ;конец процедуры
24           end     main                ;конец программы

```

Таким образом, можно считать, что мы разобрались со стандартными способами вызова ассемблерных процедур из программ на Pascal и передачи им аргументов. Эти способы будут работать всегда, но компилятор может предоставлять и более удобные средства. Их мы рассматривать не будем, так как читатель, наверное, уже понимает, что они будут сводиться, в конечном счете, к разобранной нами процедуре. Остались открытыми два вопроса:

1. Как быть с передачей данных остальных типов Pascal – ведь мы рассмотрели только данные размером в байт и слово?
2. Как возвратить значение в программу на Pascal?

Что касается ответа на первый вопрос, то необходимо вспомнить о том, что в языке Pascal существуют два способа передачи аргументов в процедуру: по ссылке и по значению.

Тип аргументов, передаваемых по *ссылке*, совпадает с типом ассемблера *dword* и с типом *pointer* в Pascal. По сути, это указатель из четырех байт на некоторый объект. Структура указателя обычна: два младших байта – смещение, два старших байта – значение сегментной составляющей адреса. С помощью такого указателя в программу на ассемблере передаются адреса следующих объектов:

- всех аргументов, объявленных при описании в программе на Pascal как *var*, независимо от их типа;
- аргументов *pointer* и *longint*;
- строк *string*;
- множеств;
- массивов и записей, имеющих размер более четырех байтов.

Аргументы по *значению* передаются следующим образом:

- для типов *char* и *byte* – как байт;
- для типа *boolean* – как байт со значением 0 или 1;
- для перечисляемых типов со значением 0...255 – как байт; более 255 – как два байта;

- для типов `integer` и `word` — как два байта (слово);
- для типа `real` — как шесть байтов (три слова);
- массивы и записи, длина которых не превышает четырех байтов, передаются «как есть».

Заметим, что аргументы таких типов, как `single`, `double`, `extended` и `comp`, передаются через стек сопроцессора.

Что касается ответа на второй вопрос, то мы рассмотрим его на конкретном примере листингов 14.17 и 14.18. Помните, что мы рассматриваем вызов из программы на Pascal внешней процедуры на ассемблере. Понятно, что вызов ради вызова вряд ли нужен — вызываемая процедура должна иметь возможность вернуть данные в вызывающую программу. Поэтому такую вызываемую процедуру правильно рассматривать как функцию. В связке Pascal—ассемблер для того, чтобы возвратить результат, процедура на ассемблере должна поместить его значение в строго определенное место (табл. 14.2).

Таблица 14.2. Возврат результата из процедуры на ассемблере в программу на Pascal

| Тип возвращаемого значения | Куда записать результат? |
|----------------------------|-------------------------------------|
| Байт | al |
| Слово | ax |
| Двойное слово | dx:ax (старшее слово:младшее слово) |
| Указатель | dx:ax (сегмент:смещение) |

В листинге 14.17 приведен текст вызывающего модуля на Pascal, а в листинге 14.18 — код вызываемого модуля на ассемблере. Программа на Pascal инициализирует две переменные `value1` и `value2`, после чего вызывает функцию на ассемблере `AddAsm` для их сложения. Результат возвращается в программу на Pascal и присваивается переменной `rez`.

Листинг 14.17. Вызывающая программа на Pascal

```
{prg14_17.pas}
program prg14101;
{внешние объявления}
function AddAsm:word; external;
{$L prg14_18.obj}
var
  value1: word; {здесь как внешние}
  value2: word;
  rez: word;
begin
  value1:=2;
  value2:=3;
{вызов функции}
  rez:=AddAsm;
  writeln("Результат: ",rez);
end.
```

Листинг 14.18. Вызываемая процедура на ассемблере

```
;prg14_18.asm
MASM
MODEL small
data segment word public;сегмент данных
    ;объявление внешних переменных
    extrn value1:WORD
    extrn value2:WORD
data ends ;конец сегмента данных
.code
    assumeds:data      ;привязка ds к сегменту
                        ;данных программы на Pascal
main:
AddAsm proc near
PUBLIC AddAsm ;внешняя
    mov cx,ds:value1 ;value1 в cx
    mov dx,ds:value2 ; value2 в dx
    add cx,dx ;сложение
    mov ax,cx ;результат в ax, так как – слово
    ret ;возврат из функции
AddAsm endp ;конец функции
end main ;конец программы
```

Как вы, наверное, обратили внимание, здесь была использована еще одна возможность доступа к разделяемым данным — использование сегментов типа `public` (см. урок 5). Совместное использование сегментов данных стало возможным благодаря тому, что компилятор Pascal создает внутреннее представление программы в виде сегментов, как и положено для программ, выполняющихся на микропроцессоре Intel. Сегмент данных в этом представлении тоже имеет название `data`, и директива `segment` для него выглядит так: `data segment word public` и т. д.

Команды `enter` и `leave`

Учитывая важность проблемы организации межмодульных связей, в систему команд микропроцессора были введены специальные команды `enter` и `leave`. Их использование позволяет облегчить написание кода пролога и эпилога в процедурах ассемблера, например:

```
;старый код пролога:
    push bp
    mov bp,sp
;новый код пролога:
    enter 0,0
;.... ...
;старый код эпилога:
    mov sp,bp
    pop bp
;новый код эпилога:
    leave
```

Транслятор ассемблера предоставляет средства в виде директив, которые еще больше упрощают работу программиста по формированию кодов пролога

и эпилога. Одной из них является директива `arg`. Другие директивы мы рассмотрим далее. Применение этих директив обсудим на конкретном примере. Для экономии места мы не будем вводить для этого новый пример, а возьмем готовый. На уроке 18 рассматриваются достаточно сложные программы, которые используют обсуждаемые здесь средства. Поэтому возьмем для примера одну из них (см. листинг 18.7). В листинге 18.7 производится обращение к процедуре `WindowProc`. При этом ей в стеке передается ряд параметров. Кроме этого, процедура использует локальные переменные. Заголовок и конец процедуры выглядят следующим образом:

```
;-----WindowProc-----  
WindowProc proc  
    arg    @@hwnd:DWORD, @emes:DWORD, @@wparam:DWORD, @@lparam:DWORD  
    uses   ebx,edi,esi,ebx ; эти регистры обязательно должны сохраняться  
    local  @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD  
....  
exit_wndproc:  
    ret  
WindowProc endp
```

В этом фрагменте новыми средствами ассемблера для нас являются директивы `uses` и `local`.

Директива `uses` содержит список регистров (см. рис. 10.3). Ее использование заставляет транслятор генерировать код для сохранения в стеке этих регистров при входе в процедуру и их восстановления при выходе из нее.

Директива `local` (см. рис. 10.3) позволяет использовать локальные переменные в процедуре. Эти переменные должны быть перечислены в список_аргументов вместе с их типами. Информацию о количестве и типах переменных транслятор использует для формирования соответствующего программного кода.

Подвернем трансляции исходный текст программы листинга 18.7 и откроем для просмотра файл `prg18_3.1st`. Посмотрим на результат применения этих директив:

```
;-----WindowProc-----  
0000012D WindowProc proc  
    arg    @@hwnd:DWORD, @emes:DWORD, @@wparam:DWORD, @@lparam:DWORD  
    uses   ebx,edi,esi  
    local  @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD  
0000012D C8 000C 00 ENTERD 0000Ch,0  
00000131 53    PUSH   ebx  
00000132 57    PUSH   edi  
00000133 56    PUSH   esi  
00000134 83 7D 0C 02 cmp    @emes,WM_DESTROY  
....  
000002CF      exit_wndproc:  
000002CF 5E    POP    esi  
000002D0 5F    POP    edi  
000002D1 5B    POP    ebx  
000002D2 C9    LEAVED  
000002D3 C20010RET 00010h  
000002D6      WindowProc endp
```

Видно, что транслятор хорошо «поработал» над кодами входа в процедуру и выхода из нее. Директива `uses ebx, edi, esi` заставляет транслятор генерировать команды `push` и `pop` для сохранения/восстановления регистров `ebx`, `edi` и `esi`. А какое влияние на формирование кода входа в процедуру и выхода из нее оказывают директивы `arg` и `local`? Чтобы разобраться с этим, подвергнем трансляции три варианта программы листинга 18.7, в каждом из которых закомментируем определенные строки:

- закомментируем строку с директивой `arg`. Фрагмент листинга будет выглядеть:

WindowProc

```
0000012D WindowProc proc
;arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses ebx,edi,esi ;эти регистры обязательно должны сохраняться
local @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD
0000012D C8 000C 00      ENTERD 0000Ch,0
00000131 53              PUSH   ebx
00000132 57              PUSH   edi
00000133 56              PUSH   esi
00000134 83 3D 00000000 02  cmp    @@mes,WM_DESTROY
;... ...
000002F9             exit_wndproc:
000002F9 5E              POP    esi
000002FA 5F              POP    edi
000002FB 5B              POP    ebx
000002FC C9              LEAVED
000002FD C3              RET    00000h
000002FE             WindowProc endp
```

- закомментируем строку с директивой `local`. Фрагмент листинга будет выглядеть:

WindowProc

```
0000012D             WindowProc proc  
    arg    @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD  
    ;uses ebx,edi,esi           ;эти регистры обязательно должны сохраняться  
    ;local @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD  
0000012D  C8 0000 00      ENTERD 00000h,0  
00000131  83 7D 0C 02    cmp    @@mes,WM_DESTROY  
;... . . .  
000002E6             exit wndproc:  
000002E6  C9              LEAVED  
000002E7  C2 0010          RET    00010h  
000002EA             WindowProc endp
```

- закомментируем строки с директивами `local` и `arg`. Фрагмент листинга будет выглядеть:

WindowProc

```
0000012D      WindowProc    proc  
    ;arg @hwnd:DWORD, @emes:DWORD, @0wparam:DWORD, @0lparam:DWORD  
    uses ebx,edi,esi    ;эти регистры обязательно должны сохраняться  
    ;local @0hdc:DWORD,@0hbrush:DWORD,@0hbit:DWORD  
0000012D 53      PUSH  ebx
```

```

0000012E 7          PUSH  edi
0000012F 56         PUSH  esi
00000130 83 3D 00000000 02 cmp   @@mes,WM_DESTROY
**Error** prg19_3.asm(209) PROCBEG(4) Undefined symbol: @@mes
*Warning* prg19_3.asm(209) PROCBEG(4) Argument needs type override
;.....
0000030F           exit_wndproc:
0000030F 5E          POP   esi
00000310 5F          POP   edi
00000311 5B          POP   ebx
00000312 C3          RET   00000h
00000313           WindowProc endp

```

Проанализируем эти фрагменты.

Применение директив `arg` и `local` приводит к генерации команд `enter` и `leave` при входе и выходе из процедуры. Поочередное комментирование этих директив показывает, что они влияют только на формирование операнда команды `enter`. При использовании `local` он равен числу байт, необходимых для размещения в стеке локальных переменных. Это так называемый *кадр стека*. Если строку с директивой `local` закомментировать, то команда `enter` все равно формируется, но с нулевым значением первого операнда. Это говорит о том, что пролог процедуры формируется в любом случае, но директива `local` позволяет еще и сформировать кадр стека для хранения локальных переменных процедуры. Соответственно, во всех вариантах генерации команды `enter` в конце процедуры формируется команда `leave`.

Теперь посмотрим на то, какое влияние оказывается директивой `arg` на формирование команды `ret` в процедуре `WindowProc`. Из анализа четырех приведенных выше вариантов фрагмента листинга видно, что в некоторых из них транслятор вычисляет суммарный размер аргументов, передаваемых в процедуру, и помещает это значение в качестве операнда команды `ret`. Это производится в тех случаях, когда директива `arg` не закомментирована. Отсюда следует вывод о прямом влиянии директивы `arg` на operand команды `ret`. Ненулевое значение операнда команды `ret` приводит к изменению значения регистра `sp\esp` — в нашем случае это означает очистку стека от аргументов, переданных в процедуру.

Отметим, что эти средства можно использовать не только для связи Pascal—ассемблер, но и для организации других межъязыковых связей, в том числе ассемблер—ассемблер.

Связь С—ассемблер

Общие принципы организации такой связи напоминают только что рассмотренное соединение Pascal и ассемблера. Поэтому мы коротко обсудим отличия на примере конкретных программ. Но прежде отметим, что язык C++ предоставляет дополнительные возможности связи программы с ассемблером, одновременно поддерживая традиционную организацию связи. Поэтому мы рассмотрим связь с ассемблером в стиле C как стандартную. При необходимости читатель, зная основы подобной связи, без труда разберется с нюансами дополнительных возможностей связи в стиле C++.

Нас по-прежнему интересуют три вопроса: как передать аргументы в процедуру на ассемблере, как к ним обратиться в процедуре на ассемблере и как возвратить результат?

Вначале отметим, что всегда нужно сохранять — и перед выходом из процедуры восстанавливать — содержимое регистров bp, sp, cs, ds и ss. Это делается перед вызовом процедуры. Остальные регистры нужно сохранять по необходимости, но, на мой взгляд, хорошим тоном является сохранение и последующее восстановление всех регистров, которые подвергаются изменению внутри процедуры.

Передача аргументов в процедуру на ассемблере из программы на С осуществляется также через стек (рис. 14.3), но порядок их размещения в стеке является обратным тому, что рассмотрен выше для связи Pascal—ассемблер. В качестве примера используем ту же задачу. После передачи управления ближнего типа процедуре на ассемблере стек имеет вид, как на рис. 14.3, a.

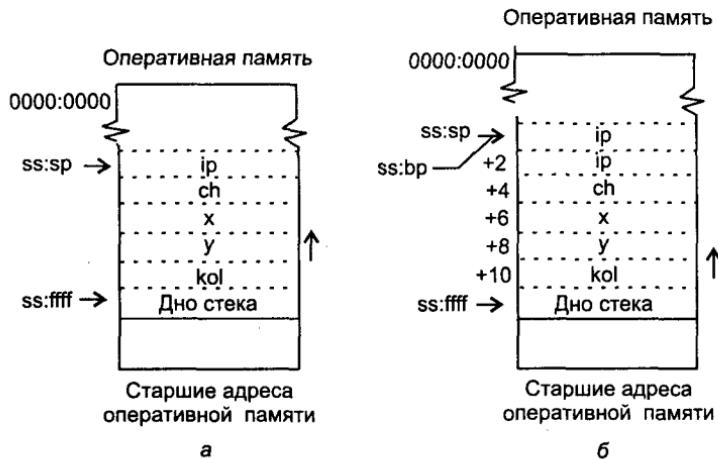


Рис. 14.3. Изменение содержимого стека при передаче управления С—ассемблер

Процедуры на ассемблере получают доступ к аргументам, переданным в стеке, посредством регистра bp. Принцип доступа — тот же, что и выше (рис. 14.3, б). Прежде всего в начало процедуры ассемблера необходимо вставить код пролога:

```
push    bp  
mov     bp, sp
```

После этого доступ к аргументам в стеке осуществляется по смещению относительно содержимого bp, например:

```
movax,[bp+4]  ;переписать значение ch  
              ;из стека в ax  
movbx,[bp+6]  ;значение x – в регистр bx
```

При организации связи С—ассемблер также можно использовать директиву arg. Это избавит нас от необходимости подсчитывать смещения в стеке для доступа к аргументам и позволит обращаться к ним просто по именам.

```

arg ch:byte;x:word;y:word;kol:word
push bp
mov bp.sp
...
mov ax,[ch]      ;переписать значение ch
                 ;из стека в ax
mov bx,[x];значение x – в регистр bx
...

```

Чтобы не повторяться, мы рассмотрим, как изменятся вызываемый и вызывающий модули (листинги 14.19 и 14.20) для связи С–ассемблер по сравнению с листингами 14.17 и 14.18.

Листинг 14.19. Вызывающий модуль на C++

```

//prg14_19.cpp
#include <studio.h>
#include <conio.h>
extern "C" void asmproc(char ch, unsigned x,
unsigned y,unsigned kol);
void main (void)
{
clrscr();
asmproc("a", 2, 3, 5);
asmproc("s", 9, 2, 7);
}

```

Листинг 14.20. Вызываемая процедура на ассемблере

```

;prg14_20.asm
MASM
MODEL small           ;модель памяти и тип кода
STACK 256
PUBLIC _asmproc        ;символ подчеркивания обязателен
.code
main:
_asmproc proc c near c:BYTE,x:WORD,y:WORD,kol:WORD
    mov dh,y          ;у-координата символа в dh
    mov dl,x          ;х-координата символа в dl
    mov ah,02h         ;номер службы BIOS
    int 10h            ;вызов прерывания BIOS
    mov ah,09h         ;номер службы BIOS
    mov cx,kol         ;kol – количество "выводов" в cx
    mov bl,07h          ;маска вывода в bl
    xor bh,bh
    mov al,c           ;c – символ в al
    int 10h            ;вызов прерывания BIOS
    ret                ;возврат из процедуры
_asmproc endp
end main

```

Что касается передачи аргументов С–ассемблер, то здесь, как видите, все довольно прозрачно. В листинге 14.20 мы используем директиву MODEL с операндом С и директиву PROC с указанием языка С. Этим мы доверяем компилятору само-

му сформировать коды пролога и эпилога, а также организовать обращение к переменным в стеке по их именам. Но при использовании конкретных программных средств организация такой связи выглядит намного проблематичней. Не в последнюю очередь это связано с тем, что компиляторы языка C/C++ разрабатывают множество фирм — в отличие от Pascal, компилятор для которого выпускает практически одна фирма Borland¹. Это обстоятельство, на мой взгляд, — основная причина сложности связи С-ассемблер, так как каждая фирма реализует ее по-своему (хотя суть и остается практически неизменной). Поэтому, как мне кажется, нет смысла рассматривать множество частных случаев, тем более, что это не является целью данной книги. Обращайтесь к документации на ваш компилятор C/C++. Опыт показывает, что достаточно хороший эффект дает применение ассемблерных вставок в программу на C/C++. Как правило, компиляторы позволяют связывать модули на C/C++ и ассемблере с использованием средств командной строки. Так как этот процесс достаточно стандартизован, есть смысл его рассмотреть. В качестве примера выберем компилятор C++ 5.0 фирмы Inprise (Borland). Типовая последовательность шагов выглядит примерно так:

- Составить текст программы на C++ (см. листинг 14.19). В этой программе — объявить процедуру `asmproc` внешней:
`extern void asmproc(char ch, unsigned x,
 unsigned y, unsigned kol);`
 - Выполнить трансляцию модуля C++ и получить объектный модуль:
`bcc -c prg14_19.cpp,`
где параметр `-c` означает, что выполняется только компиляция исходного файла, загрузочный модуль не создается. Результатом этого шага будет создание объектного модуля `prg14_19.obj`.
 - Составить текст процедуры на ассемблере (листинг 14.20), в которой объявить процедуру `asmproc` общедоступной с помощью директивы `PUBLIC`. Заметьте, что идентификатору `asmproc` предшествует символ подчеркивания — `_asmproc`. Компилятор C/C++ добавляет знак подчеркивания ко всем глобальным идентификаторам.
 - Выполнить трансляцию программы на ассемблере:
`tasm prg14_20...`
 - Выполнить объединение объектных модулей:
`bcc -ms prg14_19.obj prg14_20.obj,`
исполняемому модулю будет присвоено имя `prg14_19.exe`. Параметр `-ms` определяет модель памяти.
- Компилятор Borland C++ предоставляет другую возможность для получения загрузочного модуля. Скопируйте файл `tasm.exe` в директорий ..\bin пакета Borland C++. Запустите исходные файлы на трансляцию командной строкой вида:
- `bcc prg14_19.cpp prg14_20.asm`

¹ Известен также Microsoft Pascal. Язык Object Pascal также пользуется популярностью среди программистов для Macintosh (собственно, Object Pascal и разрабатывался для компьютеров Apple). — Примеч. ред.

В результате будет получен файл prg14_19.exe. Компилятор Borland C++ всю работу организовал сам: обработал файл prg14_19.cpp; вызвал транслятор tasm.exe, который выполнил трансляцию prg14_20.asm; передал компоновщику объектные модули prg14_19.obj и prg14_20.obj, — в результате его работы был получен загрузочный модуль prg14_19.exe.

Как возвратить результат в программу на С из процедуры на ассемблере? Для этого существуют стандартные соглашения (табл. 14.3). Перед возвратом управления в программу на С в программе на ассемблере необходимо поместить результат или сформировать указатель в указанных регистрах. Для иллюстрации работы с функцией С, текст которой написан на ассемблере, рассмотрим листинги 14.21 и 14.22. В них функция, написанная на ассемблере, подсчитывает сумму элементов массива. В функцию передаются адрес массива и его длина. Результат суммы элементов массива возвращается обратно в вызывающую программу на С.

Таблица 14.3. Возврат аргументов из процедуры на ассемблере в программу на С/С++

| Тип возвращаемого значения (C++) | Куда поместить результат? |
|----------------------------------|---------------------------|
| unsigned char | ax |
| char | ax |
| enum | ax |
| unsigned short | ax |
| short | ax |
| unsigned int | ax |
| int | ax |
| unsigned long | dx:ax |
| long | dx:ax |
| указатель near | ax |
| указатель far | dx:ax |

Листинг 14.21. Вызывающий модуль на С/С++

```
/*prg14_21.c*/
#include <stdio.h>
extern int sum_asm(int massiv[],int count);
main()
{
    int mas[5]={1,2,3,4,5};
    int len=5;
    int sum;
    sum=sum_asm(mas,len);
    printf("%d\n",sum);
    return(0);
}
```

Листинг 14.22. Вызываемая процедура на ассемблере

```
:prg14_22.asm
MASM
MODEL    small
.stack   100h
.code
    public _sum_asm
_sum_asm proc c near adr_mas:word,len_mas:word
    mov ax,0
    mov cx,len_mas ;длину массива - в cx
    mov si,adr_mas ;адрес массива - в si
    add ax,[si]      ;сложение аккумулятора с элементом массива
    add si,2          ;адресовать следующий элемент массива
    loop cyc1
    ret ;возврат из функции, результат - в ax
_sum_asm endp
end
```

Обратите внимание, что листинг 14.19 содержит текст исходного файла с расширением .cpp, а листинг 14.21 — с расширением .c. Соответственно, сами исходные тексты в части организации межмодульного взаимодействия также различаются.

Таким образом, мы рассмотрели связь модулей на языках высокого уровня с модулями на ассемблере. Это обсуждение было несколько ограниченно; его можно рассматривать как введение (хотя и достаточно подробное) в предмет. Можно было уделить данному вопросу гораздо больше внимания. Но главное — мы разобрались с принципами. Теперь дело за малым: взять литературу — лучше документацию для конкретного языка высокого уровня — и разбираться с тем, как в языке разработчики реализовали связь с ассемблером. Затем следует понять техническое осуществление этой связи, используя особенности и настройки конкретной среды программирования.

Подведем некоторые итоги:

- Язык ассемблера содержит достаточно мощные средства поддержки структурного программирования. В языке ассемблера эта технология поддерживается в основном с помощью механизма процедур и частично с использованием макрокоманд.
- Гибкость интерфейса между процедурами достигается за счет разнообразия вариантов передачи аргументов в процедуру и возвращения результатов. Для этого могут использоваться регистры, общие области памяти, стек, директивы `extrn` и `public`.
- Средства TASM поддерживают связи между языками. Ключевой момент при этом — организация обмена данными. Обмен данными между процедурами на языках высокого уровня и ассемблера производится через стек. Для доступа к аргументам используются регистр `bp` или (что более удобно) директива `arg`.

- Можно доверить компилятору самому формировать коды пролога и эпилога, указав язык в директиве `MODEL`. Кроме того, указание языка позволяет задействовать символические имена аргументов, переданных процедуре в стеке, вместо прямого использования регистра `bp` для доступа к ним. Тем самым повышаются мобильность разрабатываемых вами программ и устойчивость их к ошибкам.
- Для возврата результата в программу на языке высокого уровня необходимо использовать конкретные регистры. Через них можно передать как сами данные, так и указатели.

- Понятие прерывания
 - Классификация прерываний для микропроцессоров Intel
 - Характеристика аппаратных и программных средств обработки прерываний
 - Программируемый контроллер прерываний i8259A
 - Программирование контроллера i8259A
 - Реальный режим работы микропроцессора
 - Обработка прерываний в реальном режиме
-
-

На предыдущем уроке мы закончили рассмотрение всех основных конструкций языка ассемблера и базовых приемов программирования на нем. Обсуждение собственно языковых конструкций невольно сопровождалось выяснением тех или иных особенностей микропроцессора. Это естественно, так как язык ассемблера является символическим представлением машинного языка, который различен для разных компьютеров. Поэтому нет смысла рассматривать язык ассемблера в отрыве от принципов работы микропроцессора и наоборот. Заключительные три урока мы посвятим оставшимся за кадром, но довольно принципиальным вопросам работы микропроцессора. Тогда мы сможем получить полную картину того, что происходит в компьютере на самом низком уровне. Понимание сути этих процессов, возможно, подвигнет вас к поиску оптимальных решений при разработке программ.

Нажимая на клавиши клавиатуры, задумывались ли вы над тем, что в это время происходит в компьютере, как он узнает о том, что была нажата та или иная клавиша? Или, к примеру, каким образом ведется отсчет времени в компьютере? Ведь каждый сигнал от различных устройств компьютера, чтобы стать «осознанным», должен быть обработан какой-то программой. А выполнять программы может только микропроцессор. Но он один в компьютере, следовательно, в каждый конкретный момент времени микропроцессор может быть занят выполнением только одной программы. Как же ему узнавать о нажатиях клавиш и других сигналах, постоянно возникающих во время работы компьютера, если он выполняет некоторую программу?

Возможным решением здесь может быть, например, периодическая остановка текущей программы и выполнение других программ, производящих опрос устройств компьютера и, в свою очередь, запускающих необходимые программы для обслуживания этих устройств. Это далеко не оптимальный путь, значительно снижающий производительность компьютера. Другой возможный подход к обслуживанию устройств — создание системной очереди на обслуживание. Этот подход предполагает некую очередь, в которую «выстраиваются» запросы на обслуживание от устройств. Микропроцессор периодически просматривает эту очередь и выполняет обслуживание запросов в ней. Этот вариант, хотя и лучше предыдущего, но тоже не оптимальный. В современных микропроцессорах, каковыми являются микропроцессоры фирмы Intel, принят подход, основанный на понятии *прерывания*. Прерывание — инициируемый определенным образом процесс, временно переключающий микропроцессор на выполнение другой программы с последующим возобновлением выполнения прерванной программы.

Что дает использование механизма прерываний? Он позволяет обеспечить наиболее эффективное управление не только внешними устройствами, но, как мы увидим далее, и программами. Нажимая клавишу на клавиатуре, вы фактически инициируете посредством прерывания немедленный вызов программы, которая распознает нажатую клавишу, заносит ее код в буфер клавиатуры, откуда он в дальнейшем считывается некоторой другой программой или операционной системой. На время такой обработки микропроцессор прекращает выполнение некоторой программы и переключается на так называемую *процедуру обработки прерывания*. После того как данная процедура выполнит необходимые действия, прерванная программа продолжит выполнение с точки, где было приостановлено ее выполнение. Некоторые операционные системы используют механизм прерываний не только для обслуживания внешних устройств, но и для предоставления своих «услуг». Так, хорошо известная и до сих пор достаточно широко используемая операционная система MS-DOS взаимодействует с системными и прикладными программами преимущественно через систему прерываний.

Исходя из вышеприведенных рассуждений, можно сказать, что прерывания могут быть *внешними и внутренними*.

Внешние прерывания вызываются внешними по отношению к микропроцессору событиями. На рис. 15.1 схематически изображена подсистема прерываний компьютера на базе микропроцессора Intel.

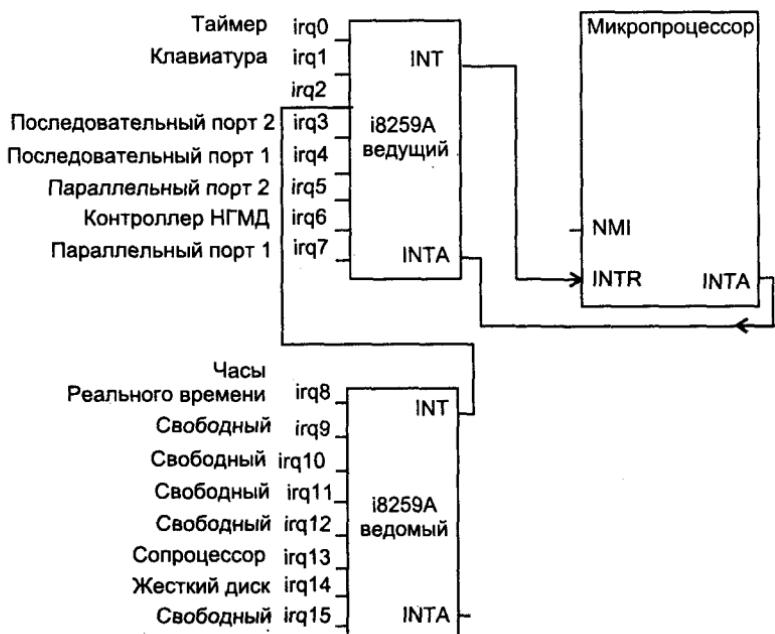


Рис. 15.1. Подсистема прерываний компьютера на базе микропроцессора Intel

На рис. 15.1 видно, что у микропроцессора есть два физических контакта — INTR и NMI. На них и формируются внешние по отношению к микропроцессору сигналы, возрастающие фронты которых извещают микропроцессор о том, что некоторое

внешнее устройство просит уделить ему внимание. Вход INTR (INTerrupt Request) предназначен для фиксации запросов от различных периферийных устройств, например таких, как системные часы, клавиатура, жесткий диск и т. д. Вход NMI (NonMaskable Interrupt) — немаскируемое прерывание. Этот вход используют для того, чтобы сообщить микропроцессору о некотором событии, требующем безотлагательной обработки, или катастрофической ошибке. Внешние прерывания относятся, естественно, к непланируемым прерываниям.

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

- ненормальное внутреннее состояние микропроцессора, возникшее при обработке некоторой команды программы. Такие события принято называть *исключительными ситуациями*, или просто *исключениями*. Этот вид прерываний отчасти также можно отнести к непланируемым;
- обработка машинной команды int xx. Этот тип прерываний называется *программным*. Это — планируемые прерывания, так как с их помощью программист обращается в нужное для него время за обслуживанием своих запросов либо к операционной системе, либо к BIOS, либо к собственным программам обработки прерываний.

Далее мы рассмотрим особенности обработки прерываний. Как уже отмечалось, микропроцессоры Intel имеют два режима работы — *реальный* и *защищенный*. В этих режимах обработка прерываний осуществляется принципиально разными методами. Поэтому на данном уроке мы дадим характеристику реального режима и рассмотрим обработку прерываний в этом режиме. На следующем уроке будет рассмотрен защищенный режим работы микропроцессора, и на последнем уроке мы рассмотрим обработку прерываний в этом режиме.

Для глубокого понимания процессов, происходящих в компьютере при осуществлении прерывания, необходимо узнать о том, какие ресурсы компьютера при этом задействуются, каковы их характеристики и принципы функционирования.

В общем случае *система прерываний* — это совокупность программных и аппаратных средств, реализующих *механизм прерываний*.

К *аппаратным средствам* системы прерываний относятся:

- выводы микропроцессора:
 - INTR — вывод для входного сигнала внешнего прерывания. На этот вход поступает выходной сигнал от микросхемы контроллера прерываний 8259A;
 - INTA — вывод микропроцессора для выходного сигнала подтверждения получения сигнала прерывания микропроцессором. Этот выходной сигнал поступает на одноименный вход INTA микросхемы контроллера прерываний 8259A;
 - NMI — вывод микропроцессора для входного сигнала немаскируемого прерывания;
- микросхема программируемого контроллера прерываний 8259A. Она предназначена для фиксирования сигналов прерываний от восьми различных внешних

устройств. В силу ее важной роли при работе всей вычислительной системы мы ее подробно рассмотрим ниже;

- внешние устройства: таймер, клавиатура, магнитные диски и т. д.

К *программным средствам* системы прерываний реального режима относятся:

- *таблица векторов прерываний*. В этой таблице в определенном формате, который зависит от режима работы микропроцессора, содержатся указатели на процедуры обработки соответствующих прерываний;

- следующие флаги в регистре флагов `flags\eflags`:

- `IF` (Interrupt Flag) — флаг прерывания. Предназначен для так называемого маскирования (запрещения) аппаратных прерываний, то есть прерываний по входу `INTR`. На обработку прерываний остальных типов флаг `IF` влияния не оказывает. Если `IF` = 1, микропроцессор обрабатывает внешние прерывания, если `IF` = 0, микропроцессор игнорирует сигналы на входе `INTR`;
- `TF` (Trace Flag) — флаг трассировки. Единичное состояние флага `TF` переводит микропроцессор в режим покомандной работы. В режиме покомандной работы после выполнения каждой машинной команды в микропроцессоре генерируется внутреннее прерывание с номером 1, и далее следуют действия в соответствии с алгоритмом обработки данного прерывания;

- машинные команды микропроцессора: `int`, `into`, `iret`, `cli`, `sti`.

Контроллер прерываний

На наш взгляд, знакомство с системой прерываний микропроцессора Intel следует начать с обсуждения организации обработки аппаратных прерываний. Как видно из рис. 15.1, центральное место в схеме обработки аппаратных прерываний занимает программируемый контроллер прерываний (ПКП), выполненный в виде специальной микросхемы i8259A. Как мы уже говорили, эта микросхема может обрабатывать запросы от восьми источников внешних прерываний. Этого явно мало, поэтому в стандартной конфигурации вычислительной системы обычно используют две последовательно соединенные микросхемы i8259A. В результате такого соединения количество возможных источников внешних прерываний возрастает до 15.

Для того чтобы разобраться с обработкой аппаратных прерываний, нам придется проникнуть внутрь микросхемы i8259A. Перечислим функции, выполняемые микросхемой контроллера прерываний:

- фиксирование запросов на обработку прерывания от восьми источников, формирование единого запроса на прерывание и подача его на вход `INTR` микропроцессора;
- формирование номера вектора прерывания и выдача его на шину данных;
- организация приоритетной обработки прерываний;
- запрещение (маскирование) прерываний с определенными номерами.

На рис. 15.2 показано схематическое представление внутренней структуры и физических выводов микросхемы i8259A.

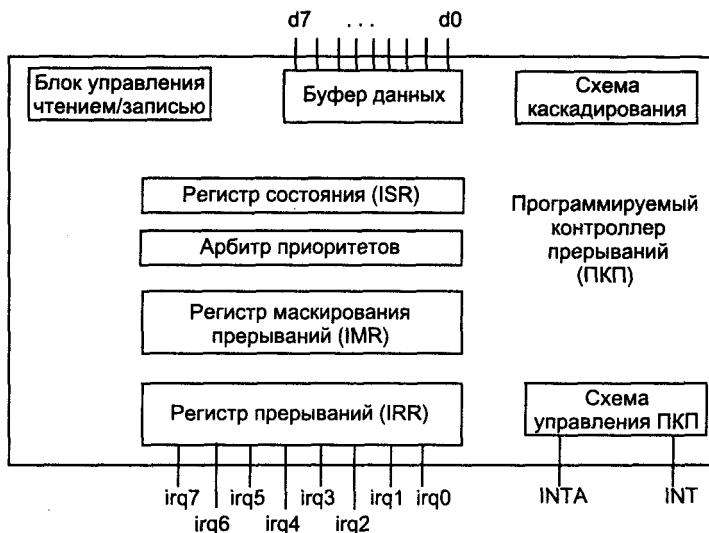


Рис. 15.2. Структурная схема и схематическое представление выводов i8259A

Рассмотрим назначение представляющих для нас интерес выводов i8259A:

d0...d7 — выводы i8259A, замыкающиеся на системную шину данных. По ним передается номер вектора прерывания и принимается управляющая информация; INT — вывод выходного сигнала запроса на прерывание, который подается на вход микропроцессора INTR;

INTA — вывод для сигнала от микропроцессора, подтверждающего факт принятия им прерывания на обслуживание;

irq0...irq7 — выводы для входных сигналов запросов на прерывания от внешних устройств.

Важное свойство данного контроллера — возможность его программирования, что позволяет достаточно гибко изменять алгоритмы обработки аппаратных прерываний. Исходя из этого, микросхема i8259A имеет два состояния:

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

Рассмотрим назначение основных структурных компонентов контроллера прерываний (см. рис. 15.2):

- *регистр запросов на прерывания IRR* (Interrupt Request Register) — восьмиразрядный регистр, фиксирующий поступление сигнала на один из входов

i8259A – `irq0...irq7`. Фиксация выражается в установке соответствующего бита в единичное состояние;

- **регистр маскирования прерываний IMR (Interrupt Mask Register)** – восьмиразрядный регистр, с помощью которого можно запретить обработку запросов на прерывания, поступающих на соответствующие входы (уровни) `irq0...irq_7`. Для запрещения (маскирования) определенных уровней прерываний необходимо установить соответствующие биты регистра IMR. Эта операция осуществляется путем программирования порта `21h`;
- **регистр обслуживаемых прерываний ISR (Interrupt Service Register)** – восьмиразрядный регистр, единичное состояние разрядов которого показывает, прерывания каких уровней обрабатываются в данный момент в микропроцессоре;
- **арбитр приоритетов PR (Priority Resolver)** – функцией данного блока является разрешение конфликта при одновременном поступлении запросов на входы `irq0...irq7`;
- **блок управления** – основной функцией данного блока является организация информационного обмена контроллера прерываний и микропроцессора через шину данных. На этот блок замыкаются как выводы `d0...d7`, так и некоторые другие (см. рис. 15.2).

Рассмотрим возможные прохождение и обработку сигнала прерывания от некоторого внешнего устройства. При этом воспользуемся структурной схемой контроллера прерываний и обозначениями на ней (см. рис. 15.2).

Допустим, на вход `irq0` поступает сигнал прерывания, что приводит к установке нулевого бита регистра IRR. Этот регистр связан с регистром маски IMR, состояния битов которого определяет, какие уровни прерываний запрещены (единичные биты) или разрешены к обработке (нулевые биты). Управление данным регистром осуществляется через порт `21h`. Таким образом, если бит 0 в IMR равен нулю, то прерывание уровня 0 разрешено. Далее сигнал поступает к арбитру приоритетов. Как мы уже отметили, функция этого блока – разрешение конфликтов при одновременном поступлении запросов на несколько уровней. Обычно самый высокий приоритет у уровня `irq_0`, и далее приоритет уменьшается с возрастанием номера уровня. Если конфликта нет, то сигнал поступает на схему управления контроллером прерываний, которая формирует сигнал на выводе `int`. Этот вывод связан со входом микропроцессора INTR. Таким образом, сигнал на входе i8259A достиг микропроцессора. Что происходит далее в микропроцессоре, мы рассмотрим ниже. Сейчас отметим только значимые для данного обсуждения моменты. Итак, при поступлении сигнала на вход INTR в микропроцессоре происходят следующие процессы:

1. Анализируется флаг IF. Если вы помните, единичное состояние этого флага говорит о том, что аппаратные прерывания разрешены, нулевое – запрещены.
2. Если прерывания запрещены, то запрос на прерывание «повисает» до момента установки IF в единицу.
3. Если прерывания разрешены, микропроцессор выполняет следующие действия:
 - сбрасывает флаг IF в ноль;

- формирует сигнал подтверждения прерывания на выводе микропроцессора INTA. Этот вывод микропроцессора замкнут на одноименный вывод микросхемы i8259A.

Таким образом, сигнал о прерывании прошел через микропроцессор и вернулся обратно в контроллер прерываний i8259A через вывод INTA. Данный вывод внутри контроллера прерываний замкнут на его схему управления, которая выполняет сразу несколько действий при поступлении этого сигнала:

1. Сбрасывает бит в регистре IRR, соответствующий уровню прерывания irq_0 .
2. Устанавливает в 1 бит 0 регистра ISR, тем самым фиксируя факт обработки прерывания уровня 0 в микропроцессоре.
3. Формирует с помощью блока управления номер вектора прерывания, значение которого формируется в буфере данных и далее поступает на выводы i8259A d0...d7. Выходы d0...d7 замкнуты на шину данных, по которой номер вектора поступает в микропроцессор. В микропроцессоре этот номер используется для вызова соответствующей процедуры обработки прерывания.

На данном этапе обработки прерывания, после того как номер прерывания пошине данных поступил в микропроцессор, последнему стало известно все об источнике прерывания. Далее микропроцессор осуществляет процедуру обработки прерывания. Если в это время придет другой сигнал о прерывании того же уровня, то он будет запомнен установкой бита в IRR, и обслуживание этого прерывания будет отложено. Если приходит прерывание другого уровня, то его дальнейшая обработка зависит от приоритета, который оно имеет по отношению к уже обрабатываемым прерываниям. Если приоритет выше, то текущая процедура обработки прерывания останавливается, и вызывается процедура обработки более приоритетного прерывания.

Очень важный момент связан с процессом завершения обработки прерывания. Проблема здесь состоит в следующем. После принятия микропроцессором запроса на обслуживание прерывания в контроллере устанавливается бит в регистре ISR, номер этого бита соответствует уровню прерывания. Установка бита с данным номером блокирует все прерывания уровня, начиная с текущего, и менее приоритетные в блоке-арбитре приоритетов. Если процедура прерывания закончит свою работу, то она сама должна этот бит сбросить, иначе все прерывания этого уровня и менее приоритетные будут игнорироваться. Для осуществления такого сброса необходимо послать код 20h в порт 20h. Есть и другая возможность – установить такой режим работы микросхемы i8259A, когда сброс этого бита будет производиться автоматически. Тонкий момент заключается в том, что происходит такой автоматический сброс будет одновременно с приходом сигнала INTA (то есть извещения о том, что запрос на обработку прерывания принят к обработке микропроцессором). Недостаток автоматического сброса в том, что существует вероятность прихода прерывания того же уровня, который уже обрабатывается в данный момент микропроцессором. В этом случае процедура обработки прерывания должна обладать свойством реентерабельности, то есть допускать повторное обращение к себе до завершения обработки предыдущего обращения. Для того чтобы процедура была реентерабельной, она должна иметь специфическую структуру, в частности, для каждого сеанса обращения к ней создается своя область для хранения переменных и значений регистров, а исполня-

емая часть процедуры находится в оперативной памяти только в одном экземпляре. Иногда может потребоваться подобный автоматический сброс, но надежнее и проще, конечно, контролировать этот процесс и самостоятельно сбрасывать бит в ISR. Это можно сделать либо в конце работы процедуры, либо в том месте процедуры, начиная с которого можно разрешить рекурсивный вызов данной процедуры, будучи уверенным в том, что она не разрушит никаких данных и работу программы в целом.

Другой не менее интересный момент заключается в том, что микропроцессор при принятии к обработке запроса на прерывание сбросил флаг IF в ноль, тем самым запретив все последующие аппаратные прерывания. Этим обстоятельством программист может пользоваться по своему усмотрению. Вы, конечно, помните, что все запросы на прерывания с приоритетом, равным текущему или меньшим, будут запрещены в любом случае, — это обусловлено логикой работы контроллера i8259A. Поэтому программист должен решить, насколько его замыслам могут помешать запросы на более приоритетные прерывания. Если это некритично, то лучше сразу, в начале процедуры обработки прерывания установить флаг IF в единицу. В большинстве случаев эту операцию нужно делать как можно раньше. Для установки флага IF в единицу в системе команд микропроцессора есть специальная команда, не имеющая операндов:

sti — разрешить аппаратные прерывания.

Наиболее наглядный пример, показывающий важность своевременной установки IF, связан с отсчетом времени. Если вы не знакомы с тем, как ведется учет времени в компьютере, то уделим этому немного внимания. Как после включения компьютер определяет текущее время суток или как он запоминает информацию о своей конфигурации после выключения? Все дело в том, что компьютер имеет небольшую энергонезависимую память, которая питается от аккумулятора и не зависит от подключения к электросети. Конструктивно эта память выполнена на специальном типе полупроводниковых элементов с так называемой CMOS-структурой (Complementar Metal Oxide Semiconductor — комплементарная МОП-структура). Особенность таких элементов памяти — в их пониженной по сравнению с обычными микросхемами потребляемой мощности (при этом они являются и более медленными, что в данном случае непринципиально). Аккумулятор кроме CMOS-памяти питает еще и микросхему системных часов, в функции которой входит отсчет текущих даты и времени суток. Таким образом, текущие значения даты и времени постоянно хранятся в CMOS-памяти и поддерживаются в актуальном состоянии даже после выключения компьютера. Кроме того, в CMOS-памяти хранится некоторая другая информация, в частности, о конфигурации компьютера. Во время загрузки компьютера дата и времячитываются в область данных BIOS. Дальнейший отсчет времени, после загрузки системы, ведется уже с помощью системного таймера — другой микросхемы на системной плате, в функции которой входит регулярно, примерно 18,2 раза в секунду, генерировать сигнал, который в качестве прерывания подается на уровень i₁q0 контроллера прерываний i8259A. Во время работы компьютера соответствующая программа BIOS обрабатывает прерывание данного уровня и ведет счет времени. Если терять такты по этому входу, то фактическое время на часах будет отставать, и поэтому в большин-

стве случаев в обработчиках прерываний есть смысл как можно раньше выдавать команду `sti`.

Программирование контроллера прерываний i8259A

Большая популярность применения этой микросхемы в качестве диспетчера аппаратных прерываний в компьютерах на базе микропроцессоров Intel объясняется наличием большого количества различных режимов ее работы, что позволяет сделать подсистему прерываний достаточно гибкой и эффективной. Действительно, если посмотреть на развитие аппаратной части компьютеров, начиная, например, с i8088/8086, то видно, что менялись самые разные компоненты, но подсистема прерываний, основанная на микросхеме i8259A, так и осталась неизменной.

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

1. **FNM** (Fully Nested Mode) — режим *вложенных прерываний*. В этом режиме каждому входу (уровню) `irq0...irq7` присваивается фиксированное значение приоритета, причем уровень `irq0` имеет наивысший приоритет, а `irq7` — наименьший. Приоритетность прерываний определяет их право на прерывание обработки менее приоритетного прерывания более приоритетным (при условии, конечно, что `IF = 1`).
2. **ARM** (Automatic Rotation Mode) — режим *циклической обработки прерываний*. В этом режиме значения приоритетов уровней прерываний также линейно упорядочены, но уже не фиксированным образом, а изменяются после обработки очередного прерывания по следующему принципу: значению приоритета последнего обслуженного прерывания присваивается наименьшее значение. Следующий по порядку уровень прерывания получает наивысшее значение, и поэтому при одновременном приходе запросов на прерывания от нескольких источников преимущество будет иметь этот уровень. Это дает возможность обеспечить «равноправие» при обработке прерываний.
3. **SRM** (Specific Rotation Mode) — режим *адресуемых приоритетов*. Этот режим можно рассматривать как вариант режима **ARM**. В режиме **SRM** программист или система самостоятельно могут назначить уровень прерывания с наивысшим приоритетом.
4. **PM** (Polling Mode) — режим *опроса*. Этот режим запрещает контроллеру автоматически прерывать работу микропроцессора при появлении прерывания от некоторого внешнего устройства. Для того чтобы микропроцессор смог узнать о наличии того или иного запроса на прерывание, он должен сам обратиться к i8259A для получения содержимого `IRR`, проанализировать его и далее действовать по своему алгоритму. Данный режим моделирует так называемую *опросную дисциплину* обработки прерываний. Мы упоминали о ней в начале урока. Согласно этому подходу, инициатором обработки прерывания становится

не само прерывание, как при *векторной дисциплине*, а микропроцессор, причем в определяемые им (точнее, операционной системой, выполняемой на нем) моменты времени.

Программирование контроллера прерываний осуществляется через адресное пространство ввода-вывода посредством двух 8-битовых портов с адресами 20h и 21h. Управление контроллером осуществляется путем посылки в определенной последовательности в эти порты специальных приказов двух типов:

- ICW (Initialization Control Word) — *управляющее слово инициализации*. Всего имеются четыре таких слова с жесткой внутренней структурой — ICW1...ICW4. Эти слова предназначены для задания режима работы контроллера. Количество этих слов (4) определено количеством режимов (см. выше).
- OCW (Operation Control Word) — *операционное управляющее слово*. Таких слов всего три, и они несут информационную нагрузку для определенных выше режимов работы контроллера прерываний. Обычно их обозначают OCW1...OCW3.

Как вы уже, наверное, успели понять, процесс программирования контроллера жестко регламентирован. Поэтому рассмотрим вначале формат приказов управления, а затем их практическое применение.

ICW1 — определить особенности последовательности приказов

Состояние битов этого приказа (табл. 15.1) определяет особенности в последовательности приказов при инициализации контроллера. Данный приказ посыпается в порт 20h

Таблица 15.1. Формат приказа ICW1

| Биты ICW1 | Назначение и содержимое |
|-----------|---|
| 0 | 1 — управляющее слово ICW4 будет присутствовать в данной последовательности приказов |
| 1 | 0 — каскадное подключение i8259A (ICW3 будет в последовательности); 1 — одиночное подключение i8259A (ICW3 не будет) |
| 2 | 0 — не используется |
| 3 | 0 — прерывание по перепаду сигнала |
| 4 | 1 — признак ICW1 |
| 5 | 0 — не используется |
| 6 | 0 — не используется |
| 7 | 0 — не используется |

ICW2 — определение базового адреса

Настало время прояснить еще один принципиальный момент, до этого времени сознательно замалчиваемый. Он связан с принципом определения числового диапазона адресов векторов прерываний для аппаратных прерываний, замкнутых

на контроллер прерываний. В реальном режиме работы микропроцессора для хранения указателей (векторов) на процедуры-обработчики прерываний используется специальная область памяти — таблица векторов прерываний. Эта таблица начинается с нулевого адреса оперативной памяти и занимает 1 Кбайт. Среди векторов есть, конечно, и вектора, указывающие на процедуры-обработчики тех прерываний, которые замкнуты на контроллер. Эти вектора располагаются в таблице последовательно, одной группой, и их нумерация начинается с некоторого номера вектора, называемого **базовым**. Приказ ICW2 (табл. 15.2) позволяет задать номер этого базового вектора для контроллера прерываний в соответствии с тем номером, который назначен соответствующему вектору в таблице векторов прерываний. В реальном режиме работы микропроцессора BIOS в процессе начальной загрузки системы инициализируется ведущий контроллер значением 08h, а ведомый — значением 70h. Теперь понятно, почему обработчику прерываний от таймера соответствует номер вектора 08h в таблице векторов прерываний, хотя физически он замкнут на уровень 0 контроллера i8259A. При желании мы вполне можем изменить значение базового номера на любой не используемый в системе номер, к примеру — 90h. Также следует учитывать, что некорректная установка нового номера этим приказом может полностью нарушить работу всей системы. Данный приказ посыпается в порт 21h.

Таблица 15.2. Формат приказа ICW2

| Биты ICW2 | Назначение и содержимое |
|-----------|---|
| 0 | 0 — не используется |
| 1 | 0 — не используется |
| 2 | 0 — не используется |
| 3 | Бит для задания номера базового вектора |
| 4 | Бит для задания номера базового вектора |
| 5 | Бит для задания номера базового вектора |
| 6 | Бит для задания номера базового вектора |
| 7 | Бит для задания номера базового вектора |

Как видно, для задания номера базового вектора используются биты с 3 по 7 приказа ICW2. Объяснить это можно тем, что на контроллер замыкаются 8 источников прерываний. Выше мы отметили, что номера векторов, соответствующих прерываниям, замкнутых на контроллер, имеют последовательные номера, начиная с базового. Так, для контроллера, инициализированного значением базового номера 08h, номера векторов в таблице векторов прерываний будут 08h, 09h, 0ah, 0bh и т. д. Отсюда и получается, что для задания базового номера биты 0...2 использовать нельзя, так как они применяются для формирования адресов векторов прерываний следующих после базового уровней.

ICW3 — связь контроллеров

Этот приказ предназначен для связи контроллеров в системе с несколькими контроллерами прерываний.

Вариант работы с одной микросхемой i8259A, позволяющий обрабатывать запросы от 8 источников, использовался в ранних системах на базе микропроцессоров i8088/86 (в архитектуре XT). Но i8259A позволяет организовать так называемое *каскадное соединение* этих микросхем, при котором выход INT одной микросхемы подается на вход одного из уровней irq другой микросхемы (см. рис. 15.1). Это позволяет организовать обработку запросов от большего числа источников. При этом один контроллер является *ведущим*, а другой — *ведомым* (тот, который подключен ко входу irq ведущего). Ниже мы разберемся с каскадированием более подробно. Сейчас отметим, что формат приказа ICW3 зависит от того, какой контроллер инициализируется — ведущий (табл. 15.3) или ведомый (табл. 15.4). При инициализации ведущего контроллера ICW3 сообщает, к каким его входам irq подсоединенны ведомые контроллеры. Соответственно, при инициализации ведомого контроллера нужна другая форма этого приказа, которая несет информацию о том, к какому входу ведущего подключен данный ведомый контроллер. Приказ ICW3 посыпается в порт 21h.

Таблица 15.3. Формат приказа ICW3 (для ведущего контроллера)

| Биты ICW3 | Назначение и содержимое |
|-----------|--|
| 0 | 1 — если ко входу $irq0$ подключен ведомый контроллер; 0 — если ко входу $irq0$ подключено внешнее устройство |
| 1 | 1 — если ко входу $irq1$ подключен ведомый контроллер; 0 — если ко входу $irq1$ подключено внешнее устройство |
| 2 | 1 — если ко входу $irq2$ подключен ведомый контроллер; 0 — если ко входу $irq2$ подключено внешнее устройство |
| 3 | 1 — если ко входу $irq3$ подключен ведомый контроллер; 0 — если ко входу $irq3$ подключено внешнее устройство |
| 4 | 1 — если ко входу $irq4$ подключен ведомый контроллер; 0 — если ко входу $irq4$ подключено внешнее устройство |
| 5 | 1 — если ко входу $irq5$ подключен ведомый контроллер; 0 — если ко входу $irq5$ подключено внешнее устройство |
| 6 | 1 — если ко входу $irq6$ подключен ведомый контроллер; 0 — если ко входу $irq6$ подключено внешнее устройство |
| 7 | 1 — если ко входу $irq7$ подключен ведомый контроллер; 0 — если ко входу $irq7$ подключено внешнее устройство |

Таблица 15.4. Формат приказа ICW3 (для ведомого контроллера)

| Биты ICW3 | Назначение и содержимое |
|-----------|---|
| 0...3 | Код идентификации номера входа ведущего контроллера, к которому подсоединен ведомый |
| 4...7 | Всегда 0 |

ICW4 — дополнительные особенности обработки прерываний

Этот приказ (табл. 15.5) определяет дополнительные особенности обработки прерываний контроллером i8259A. Данный приказ посыпается в порт 21h.

Таблица 15.5. Формат приказа ICW4

| Биты ICW4 | Назначение и содержимое |
|------------------|--|
| 0 | Тип микропроцессора: 0 — i8080; 1 — i80x86 (Pentium) |
| 1 | Особенности обработки конца прерывания (сигнала EOI): 0 — сброс бита в ISR производит программа обработки прерывания; 1 — установка режима автоматического сброса бита в ISR (после получения сигнала INTA от микропроцессора) |
| 2 | 0 — данный контроллер ведомый; 1 — данный контроллер ведущий |
| 3 | Указывает буферизованность системной шины данных: 0 — системная шина не буферизована; 1 — системная шина буферизована |
| 4 | 0 — определяет использование специального вложенного режима |
| 5...7 | 0 |

Таким образом, приказы инициализации задают контроллеру режимы работы в условиях вложенных прерываний. Если требуется конкретизировать порядок обработки для отдельных уровней прерываний, необходимо использовать специальные операционные управляющие слова — OCW, назначение и форматы которых мы рассмотрим ниже.

OCW1 — управление регистром масок IMR

Этот приказ (табл. 15.6) предназначен для управления регистром масок IMR для маскирования прерываний конкретных уровней. Данный приказ посыпается в порт 21h.

Таблица 15.6. Формат приказа OCW1

| Биты OCW1 | Назначение и содержимое |
|------------------|---|
| 0 | 0 — разрешить прерывания уровня 0; 1 — запретить прерывание уровня 0 |
| 1...6 | Для уровней 1...6 — аналогично |
| 7 | 0 — разрешить прерывания уровня 7; 1 — запретить прерывание уровня 7 |

OCW2 — управление приоритетом

Этот приказ (табл. 15.7) используется для управления приоритетом и учета особенностей завершения обслуживания прерывания в контроллере. Он определяет выполнение следующих действий:

- сбросить бит в ISR с наибольшим приоритетом;
- сбросить бит в ISR для определенного уровня прерываний;
- установить низший приоритет для определенного уровня;
- поменять приоритеты уровней с максимальным и минимальным приоритетами;
- поменять приоритеты уровней с заданным и минимальным приоритетами.

Данный приказ посыпается в порт 20h.

Таблица 15.7. Формат приказа OCW2

| Биты OCW2 | Назначение и содержимое |
|-----------|---|
| 0...2 | 000-nnn — код уровня запроса irq для действий, определяемых разрядами 5–7 |
| 3...4 | 00 — признак OCW2 |
| 5 | 0 — режим автоматического EOI; 1 — режим неавтоматического EOI |
| 6...7 | Задают операцию (возможные сочетания с 5-м битом): 000 — обычный режим приоритетов с автоматическим EOI; 001 — сброс бита с максимальным приоритетом в ISR; 011 — сброс бита в ISR для уровня с кодом nnn; 100 — установка режима циклической смены приоритета при автоматическом EOI (см. выше); 101 — установка режима циклической смены приоритета при неавтоматическом EOI; 111 — установка режима циклической смены приоритета, но относительно бита nnn |

OCW3 — общее управление контроллером

Этот приказ (табл. 15.8) служит для общего управления контроллером. Приказ OCW3 посыпается в порт 20h.

Таблица 15.8. Формат приказа OCW3

| Биты OCW3 | Назначение и содержимое |
|-----------|---|
| 0...1 | 10 — прочитать содержимое IRR (следующей командой из порта 020h); 11 — прочитать содержимое ISR (следующей командой из порта 020h); Кстати, содержимое IMR доступно постоянно как содержимое порта 021h |
| 2 | 0 или 1 — устанавливает (или не устанавливает) режим опроса, в котором можно непосредственно опрашивать контроллер о поступивших запросах на прерывание, что полезно, если прерывания в микропроцессоре запрещены (IF = 0), или когда в системе много контроллеров прерываний |
| 3...4 | 01 — признак OCW3 |

| Биты OCW3 | Назначение и содержимое |
|-----------|--|
| 5...6 | 11 — установить режим специального маскирования, в котором все немаскированные в IMR запросы будут обрабатываться микропроцессором вне зависимости от состояния ISR; 10 — сбросить режим специального маскирования; 10 или 01 — ничего не менять |
| 7 | 0 |

Этой информации уже вполне достаточно для написания и понимания реальных программ. В качестве примера рассмотрим последовательность приказов, которые выдает BIOS при загрузке системы после включения питания (табл. 15.9). Это будет хорошей иллюстрацией к вышеупомянутым рассуждениям.

Таблица 15.9. Приказы BIOS для инициализации контроллера прерываний

| Код приказа | Вид приказа и в какой контроллер посыпается | Порт |
|-------------|--|------|
| 00010001 | ICW1 в контроллер 1 | 020h |
| 00001000 | ICW2 в контроллер 1 — адрес вектора для 000020h | 021h |
| 00000100 | ICW3 в контроллер 1 — указывает подсоединение ведомого | 021h |
| 00000001 | ICW4 в контроллер 1 — режим i80x86 | 021h |
| 00010001 | ICW1 в контроллер 2 | 0a0h |
| 01110000 | ICW2 в контроллер 2 — адрес вектора для 0001c0h | 0a1h |
| 00000010 | ICW3 в контроллер 2 — указывает индекс ведомого | 0a1h |
| 00000001 | ICW4 в контроллер 2 — режим i80x86 | 0a1h |
| 10111000 | В контроллер 1 — маска прерывания OCW1 | 021h |
| 10111101 | В контроллер 2 — маска прерывания OCW2 | 0a1h |

Каскадирование микросхем i8259A

Выше мы упоминали, что один контроллер прерываний обрабатывает запросы всего лишь от 8 источников. Компьютеры XT имели всего одну микросхему i8259A, которая обрабатывала прерывания от следующих источников:

- уровень 0 — таймера;
- уровень 1 — клавиатуры;
- уровень 2 — зарезервировано для нестандартных внешних устройств;
- уровень 3 — порта COM2;
- уровень 4 — порта COM1;
- уровень 5 — жесткого диска;
- уровень 6 — НГМД;
- уровень 7 — параллельного порта — принтера.

В компьютерах АТ (с микропроцессорами i286 и выше) в связи с возросшей номенклатурой внешних устройств восьми источников прерываний стало недостаточно. Нужно было как-то расширить этот диапазон. Здесь и пригодилась способность микросхем i8259A работать в связке, или *каскадом*. Максимально можно таким образом соединить 8 контроллеров, что позволит обработать запросы от 64 источников. В архитектуре АТ используются два контроллера, соединенные каскадом, что позволяет обрабатывать запросы от 15 источников прерываний (табл. 15.10). Один из этих контроллеров является ведущим, другой — ведомым. Выход INT ведомого контроллера замкнут на вход уровня 2 ведущего контроллера (см. рис 15.1). При рассмотрении табл. 15.10 обратите внимание на распределение приоритетов.

Таблица 15.10. Распределение и приоритеты аппаратных прерываний в архитектуре АТ

| Уровень (вход) | Контроллер | Источник прерывания | Приоритет уровня |
|-------------------|------------|---|------------------|
| irq0 | Ведущий | Таймер | 2 |
| irq1 | Ведущий | Клавиатура | 3 |
| irq2 | Ведущий | Выход INT ведомого | |
| irq8 | Ведомый | Часы реального времени (CMOS-блок) | 4 |
| irq9 | Ведомый | Вход для устройства расширения | 5 |
| irq10 | Ведомый | Вход для устройства расширения | 6 |
| irq11 | Ведомый | Вход для устройства расширения | 7 |
| irq12 | Ведомый | Вход для устройства расширения | 8 |
| irq13 | Ведомый | Ошибка сопроцессора | 9 |
| irq14 | Ведомый | Контроллер жесткого диска | 10 |
| irq15 | Ведомый | Вход для устройства расширения | 11 |
| irq3 | Ведущий | Вход для устройства расширения (последовательный порт COM2) | 12 |
| irq4 | Ведущий | Вход для устройства расширения (последовательный порт COM1) | 13 |
| irq5 | Ведущий | Вход для устройства расширения (параллельный порт LPT2) | 14 |
| irq6 | Ведущий | Контроллер гибкого диска | 15 |
| irq7 | Ведущий | Вход для устройства расширения (параллельный порт LPT1) | 16 |

Реальный режим работы микропроцессора

Для тех пользователей, которые работали с микропроцессорами i8086 или i8088, нет необходимости пояснять особенности этого режима. Относительно недавно это был единственный режим, в котором функционировала популярная

операционная система MS-DOS. Для нее был разработан большой объем программного обеспечения. Понимая все это и не желая терять рынок, фирма Intel во всех модернизациях своего микропроцессора поддерживает этот режим. В нашей книге при написании программ мы, до сего момента, также подразумевали реальный режим. Вот некоторые его характеристики:

- пространство оперативной памяти делится на сегменты по 64 Кбайт. Сегменты в памяти могут перекрываться;
- страничное преобразование адреса запрещено, то есть физический адрес равен линейному и формируется как сумма двух составляющих (см. урок 2):
 - 16-разрядного эффективного адреса, который, в свою очередь, является суммой трех составляющих: базы, смещения и индекса;
 - 20-разрядного результата сдвига содержимого конкретного сегментного регистра на 4 разряда влево;
- максимальное значение физического адреса равно 0ff fffh, то есть 1 Мбайт, но, фактически, в реальном режиме микропроцессора адресуется на 64 Кбайт больше, что следует из следующего вычисления:

ffff0 – максимальное значение сегментной части адреса, сдвинутое на 4 разряда влево;
+
0ffff – максимальное значение смещения;
 $10ffef = 1\ 114\ 096$ байт – максимальный физический адрес в реальном режиме.

Этот пример говорит о том, что в модели микропроцессоров, начиная с i286, при определенных обстоятельствах возможна адресация оперативной памяти за пределами первого мегабайта. Это обстоятельство даже использовалось последними версиями MS-DOS для размещения служебных программ в этом дополнительном сегменте памяти. Формирование значений адреса сразу за первым мегабайтом возможно и в микропроцессоре i8088/86. В нем при появлении физического адреса большего 0fffffh, например 1 000 054h, микропроцессор отбрасывает 21-й единичный бит. Происходит так называемое «заворачивание» адреса, поэтому сформированный физический адрес на шине адреса будет равен 00054h. Для того чтобы обеспечить полную эмуляцию данной особенности микропроцессора i8088/86, в моделях микропроцессоров, начиная с i80286, была предусмотрена возможность блокировки адресной линии A20 (управление тем самым 21-м битом адреса). Для обеспечения доступа к адресам оперативной памяти, лежащим за пределами первого мегабайта, необходимо специальным образом открывать эту адресную линию;

- в реальном режиме схема распределения оперативной памяти – фиксированная. Перечислим расположение некоторых из системных областей, которые потребуются нам в дальнейшем:
 - в диапазоне адресов 00000h–003ffh (первый мегабайт оперативной памяти) находится *таблица векторов прерываний* (ТВП). Она содержит 256 векторов прерываний размером 4 байта (указателей на программы обработки прерываний);

- в диапазоне адресов 00400h—006ffh сразу за таблицей векторов прерываний располагается область памяти, содержащая жестко структурированные данные, обеспечивающие работу BIOS и MS-DOS;
- с адреса 0b8000h располагается область *видеопамяти*, в которой формируется изображение, которое мы видим на экране.

Обработка прерываний в реальном режиме

Обработка прерываний (как внешних, так и внутренних) в реальном режиме микропроцессора производится в три этапа:

1. Прекращение выполнения текущей программы.
2. Переход к выполнению и выполнение программы обработки прерываний.
- 3 Возврат управления прерванной программе.

Первый этап должен обеспечить временное прекращение выполнения текущей программы таким образом, чтобы потом прерванная программа продолжила свою работу так, как будто никакого прерывания не было. Любая программа, загруженная для выполнения операционной системой, занимает свое, отдельное от других программ, место в оперативной памяти. Разделяемыми между программами ресурсами являются регистры микропроцессора, в том числе регистр флагов, поэтому их содержимое нужно сохранять. Обязательными для сохранения являются регистры cs, ip и flags\eflags, поэтому они при возникновении прерывания сохраняются микропроцессором автоматически. Пара cs : ip содержит адрес команды, с которой необходимо начать выполнение после возврата из программы обслуживания прерывания, а flags\eflags — состояние флагов после выполнения последней команды прерванной программы в момент передачи управления программе обработки прерывания. Сохранение содержимого остальных регистров должно обеспечиваться программистом в начале программы обработки прерывания до их использования. Наиболее удобным местом хранения регистров является стек. В конце первого этапа микропроцессор после включения в стек регистров flags, cs и ip сбрасывает бит флага прерываний IF в регистре flags (но при этом в стек записывается предыдущее содержимое регистра flags с еще установленным IF). Тем самым предотвращаются возможность возникновения вложенных прерываний по входу INTR и порча регистров исходной программы вследствие неконтролируемых действий со стороны программы обработки вложенного прерывания. После того как необходимые действия по сохранению контекста завершены, обработчик аппаратного прерывания может разрешить вложенные прерывания командой sti.

Набор действий по реализации второго этапа заключается в определении источника прерывания и вызова соответствующей программы обработки. В реальном режиме микропроцессора допускается от 0 до 255 источников прерываний. Количество источников прерываний ограничено размером таблицы векторов прерываний. Эта таблица выступает связующим звеном между источником прерывания и процедурой обработки. Данная таблица располагается в памяти, начиная с адреса 0. Каждый элемент таблицы векторов прерываний занимает 4 байта и имеет следующую структуру:

- 1-е слово элемента таблицы — значение смещения начала процедуры обработки прерывания (*n*) от начала кодового сегмента;

○ 2-е слово элемента таблицы — значение базового адреса сегмента, в котором находится процедура обработки прерывания.

Определить адрес, по которому находится вектор прерывания с номером n , можно следующим образом:

`смещение_элемента_таблицы_векторов_прерываний = n * 4`

Таким образом, полный размер таблицы векторов прерываний $4 * 256 = 1024$ байт.

Теперь понятно, что на втором этапе обработки прерывания микропроцессор выполняет следующие действия:

1. По номеру источника прерывания путем умножения на 4 определяет смещение в таблице векторов прерываний.
2. Помещает первые два байта по вычисленному адресу в регистр `i p`.
3. Помещает вторые два байта по вычисленному адресу в регистр `c s`.
4. Передает управление по адресу, определяемому парой `c s : i p`.

Далее выполняется сама программа обработки прерывания. Она, в свою очередь, также может быть прервана, например, поступлением запроса от более приоритетного источника. В этом случае этапы 1 и 2 будут повторены для вновь поступившего запроса.

Набор действий по реализации этапа 3 заключается в восстановлении контекста прерванной программы. Так же, как и на этапе 1, на данном последнем этапе есть действия, выполняемые микропроцессором автоматически, и действия, задаваемые программистом. Основная задача на этапе 3 — привести стек в состояние, в котором он был сразу после передачи управления данной процедуре. Для этого программист указывает необходимые действия по восстановлению регистров и очистке стека. Этот участок кода необходимо защитить от возможности искажения содержимого регистров (в результате появления аппаратного прерывания) с помощью команды `c l i`. Последние команды в процедуре обработки прерывания — `st i` и `i ret`, при обработке которых микропроцессор выполняет следующие действия:

- 1) `st i` — разрешить аппаратные прерывания по входу `INTR`;
- 2) `i ret` — извлечь последовательно три слова из стека и поместить их, соответственно, в регистры `i p`, `c s` и `flags`.

В результате этапа 3 управление возвращается очередной команде прерванной программы, которая должна была выполниться, если бы прерывания не было.

Аппаратные прерывания могут быть инициированы программно командой микропроцессора `int n`, где n — номер аппаратного прерывания в соответствии с таблицей векторов прерываний. При этом микропроцессор также сбрасывает флаг `IF`, но не вырабатывает сигнал `INTA`.

После такого обстоятельного обсуждения нам осталось рассмотреть хороший пример. Он должен показать нам ключевые моменты программирования программных и аппаратных прерываний. Выберем одно программное и одно аппаратное прерывание. Наиболее «частое» аппаратное прерывание — прерывание от таймера. Что касается программного прерывания, то основное требование при его

выборе для нашего эксперимента то, чтобы его номер не совпадал с номером какого-нибудь системного прерывания.

Программа, которую мы должны будем разработать, выполняет следующие действия: подключает новый аппаратный обработчик прерываний от таймера 08h, который на каждый 4-й сигнал в цикле выводит на экран символы (0–9). Пользовательское прерывание (новый обработчик прерывания 0ffh) вызывается после запуска прерывания от таймера. Его работа заключается в выдаче сигнала сирены циклически несколько раз. После этого пользовательское прерывание производит восстановление вектора старого обработчика прерывания от таймера и завершает свою работу вместе со всей программой.

Перед обсуждением программы нужно сделать следующее замечание. По сути, мы ставим себе цель дополнить программу обработки прерывания от таймера некоторым новым свойством. Одновременно, мы не хотим портить старый обработчик этого прерывания. Такая ситуация встречается довольно часто. Существуют различные способы сцепления системных обработчиков прерываний с пользовательскими. Прерывание от таймера 08h интересно тем, что программа его обработки предусматривает возможность того, что пользователь захочет вставить в обработчик свой код. С этой целью из системной программы обработки прерывания 08h делается вызов еще одного прерывания с номером 1ch. Это пустое прерывание, обработчик которого содержит всего одну команду iret. Таким образом, пользователь имеет возможность решить проблему сцепления своего обработчика с системным обработчиком прерывания 08h косвенно — попросту заменив вектор обработчика прерывания 1ch. Этот прием реализован в листинге 15.1.

Листинг 15.1. Обработка прерывания от таймера

```
<1>      :prg15_1.asm
<2>      MASM
<3>      MODEL small           ; модель памяти
<4>      STACK 256            ; размер стека
<5>      .486p
<6>      delay macro time
<7>      local ext,iter
<8>      ;макрос задержки
<9>      ;На входе – значение переменной задержки (в мкс)
<10>     push cx
<11>     mov cx,time
<12>     ext:
<13>     push cx
<14>     ;в cx одна мкс, это значение можно
<15>     ;поменять в зависимости от производительности процессора
<16>     mov cx,5000
<17>     iter:
<18>     loop iter
<19>     pop cx
<20>     loop ext
<21>     pop cx
<22>     endm ;конец макроса
<23>     .data
<24>     tonelow dw 2651          ;нижняя граница звучания 450 Гц
```

Листинг 15.1 (продолжение)

```
<25>    cnt  db   0          ;счетчик для выхода из программы
<26>    temp dw   ?          ;верхняя граница звучания
<27>    old_off8 dw   0      ;для хранения старых значений вектора
<28>    old_seg8 dw   0      ;сегмент и смещение
<29>    time_1ch dw   0      ;переменная для пересчета
<30>    .code
<31>    off_1ch equ  1ch*4    ;смещение вектора 1ch в ТВП
<32>    off_0ffh equ  0ffh*4   ;смещение вектора ffh в ТВП
<33>    char db   "0"        ;символ для вывода на экран
<34>    maskf db  07h        ;маска вывода символов на экран
<35>    position dw  2000     ;позиция на экране – почти центр
<36>    main proc
<37>        mov  ax,@data
<38>        mov  ds,ax
<39>        xor  ax,ax
<40>        cli               ;запрет аппаратных прерываний на время
<41>                           ;замены векторов прерываний
<42>    ;замена старого вектора 1ch на адрес new_1ch
<43>    ;настройка es на начало таблицы векторов
<44>    ;прерываний – в реальном режиме:
<45>        mov  ax,0
<46>        mov  es,ax
<47>    ;сохранить старый вектор
<48>        mov  ax,es:[off_1ch]  ;смещение старого вектора 1ch в ах
<49>        mov  old_off8,ax    ;сохранение смещения в old_off8
<50>        mov  ax,es:[off_1ch+2] ;сегмент старого вектора 1ch в ах
<51>        mov  old_seg8,ax    ;сохранение сегмента в old_seg8
<52>    ;записать новый вектор в таблицу векторов прерываний
<53>        mov  ax,offset new_1ch ;смещение нового обработчика в ах
<54>        mov  es:off_1ch,ax
<55>        push cs
<56>        pop  ax             ;настройка ах на cs
<57>        mov  es:off_1ch+2,ax  ;запись сегмента
<58>    ;инициализировать вектор пользовательского прерывания 0ffh
<59>        mov  ax,offset new_0ffh
<60>        mov  es:off_0ffh,ax   ;прерывание 0ffh
<61>        push cs
<62>        pop  ax
<63>        mov  es:off_0ffh+2,ax
<64>        sti               ;разрешение аппаратных прерываний
<65>    ;задержка, чтобы новый обработчик таймера вывел символы на экран
<66>    delay 3500
<67>    ;завершение программы
<68>        int   0ffh
<69>    exit:
<70>        mov  ax,4c00h
<71>        int   21h
<72>    main endp
<73>    new_1ch proc           ;новый обработчик прерывания от таймера
<74>    ;сохранение в стеке используемых регистров
<75>        push ax
```

```

<76>      push  bx
<77>      push  es
<78>      push  ds
<79> ;настройка ds на cs
<80>      push  cs
<81>      pop   ds
<82> ;запись в es адреса начала видеопамяти – B800:0000
<83>      mov    ax,0b800h
<84>      mov    es,ax
<85>      mov    al,char   :символ в al
<86>      mov    ah,maskf  :маску вывода ~ в ah
<87>      mov    bx,position :позицию на экране – в bx
<88>      mov    es:[bx],ax  :вывод символа в центр экрана
<89>      add    bx,2       :увеличение позиции
<90>      mov    position,bx :сохранение новой позиции
<91>      inc    char     :следующий символ
<92> ;восстановление используемых регистров:
<93>      pop    ds
<94>      pop    es
<95>      pop    bx
<96>      pop    ax
<97>      iret   :возврат из прерывания
<98> new_1ch  endp   :конец обработчика
<99> new_0ffh proc   ;новый обработчик пользовательского прерывания
<100> sirena:
<101> ;сохранение в стеке используемых регистров
<102>      push  ax
<103>      push  bx
<104> ;проверка для пересчета на 4:
<105>      test   time_1ch,03h
<106>      jnz    leave_it  ;если два правых бита не 11, то на выход,
                           ;иначе:
<107> go:
<108>      mov    ax,0B06h   ;заносим слово состояния 110110110b
<109> ;(0B6h) – выбираем второй канал порта 43h (динамик)
<110>      out   43h,ax ;в порт 43h
<111>      in    al,61h ;получим значение порта 61h в al
<112>      or    al,3    ;инициализируем динамик – подаем ток
<113>      out   61h,al ;в порт 61h
<114>      mov    cx,2083  ;количество шагов
<115> musicup:
<116> ;значение нижней границы частоты вах (1193000/2651=450 Гц).
<117> ;где 1193000 – частота динамика
<118>      mov    ax,tonelow
<119>      out   42h,al ;в порт 42h – младшее слово ax:al
<120>      mov    al,ah   ;обмен между al и ah
<121>      out   42h,al ;в порте 42h уже старшее слово ax:ah
<122>      add    tonelow,1  ;увеличение частоты
<123>      delay 1      ;задержка на 1 мкс
<124>      mov    dx,tonelow ;текущее значение частоты – в dx
<125>      mov    temp,dx  ;в temp – верхнее значение частоты
<126>      loop   musicup ;повторить цикл повышения

```

Листинг 15.1 (продолжение)

```

<127>      mov  cx,2083
<128>      musicdown:
<129>      mov  ax,temp   ;верхнее значение частоты – в ах
<130>      out  42h,al ;младший байт ах:ал в порт 42h
<131>      mov  al,ah    ;обмен между ал и ах
<132>      out  42h,al ;старший байт ах:ах в порт 42h
<133>      sub  temp,1 ;уменьшение частоты
<134>      delay 1      ;задержка на 1 мкс
<135>      loop musicdown ;повторить цикл понижения
<136>      nosound:
<137>      in   al,61h ;значение порта 61h – в ал
<138>      ;слово состояния 0fch – выключение динамика и таймера
<139>      and  al,0fch
<140>      out  61h,al ;в порт 61h
<141>      mov  dx,2651  ;для последующих циклов
<142>      mov  tonelow,dx
<143>      inc  cnt     ;инкремент количества проходов
<144>      cmp  cnt,2   ;если сирена не звучала двух
                           ;раз – повторный запуск
<145>      jne  go
<146>      leave_it:           ;выход
<147>      inc  time_1ch  ;пересчет на 4
<148>      ;восстановление используемых регистров
<149>      pop  bx
<150>      pop  ax
<151>      ;восстановление вектора прерывания от таймера
<152>      cli   ;запрет аппаратных прерываний
<153>      xor  ax,ax  ;снова настройка es на начало таблицы
<154>      mov  es,ax  ;векторов прерываний
<155>      mov  ax,old_off8 ;запись в таблицу смещения старого
<156>      mov  es:off_1ch,ax ;обработчика прерывания от таймера
<157>      mov  ax,old_seg8 ;запись сегмента
<158>      mov  es:off_1ch+2,ax
<159>      sti   ;разрешение аппаратных прерываний
<160>      iret  ;возврат из прерывания
<161>      new_0ffh endp ;конец обработчика
<162>      end  main  ;конец программы

```

Обсудим листинг 15.1. Основная процедура `main` (строки 36–72) выполняет инициализацию используемых векторов прерываний. При этом необходимо запомнить содержимое старого вектора прерывания `1ch` (строки 45–51), так как его придется восстанавливать перед завершением программы. Содержимое вектора пользовательского прерывания `0ffh` сохранять нет смысла, так как его номер выбран исходя из того, что он не используется при работе системы. При смене вектора прерывания `1ch` необходимо запретить обработку аппаратных прерываний командой `c1 i` (строка 40), так как внешние прерывания являются асинхронными и могут прийти в самый неподходящий момент, в том числе и во время смены содержимого вектора. Перед завершением работы аппаратного прерывания необходимо явно выдать сигнал `E0I`. Но в нашем случае это делать необязательно, так

как за нас это сделает системный обработчик прерывания 08h, из которого вызывается обработчик для прерывания 1ch. В строках 52–57 и 58–63 производится запись новых значений векторов 1ch и 0ffh в таблицу векторов прерываний. После того как в строке 64 командой st i будут разрешены аппаратные прерывания, на экран будут выведены символы. Эти действия выполняет новая программа обработки прерывания для вектора 1ch (строки 73–98). Эти символы будут выводиться до тех пор, пока действует программная задержка, которую мы организовали в строке 66. После этого вызывается пользовательское прерывание 0ffh, программа обработки которого (строки 99–161) отрабатывает несколько циклов генерации сигнала «сирена» (мы обсуждали эту программу на уроке 7, и теперь вы уже в состоянии оформить ее в виде макроса или процедуры). Вызов программы обработки прерывания пользователя new_0ffh осуществляется с помощью специальной команды int. Эта команда предназначена для того, чтобы пользователь сам мог инициировать вызов прерываний. Как видите, эти прерывания являются планируемыми (синхронными), так как пользователь сам определяет момент его вызова.

После написания этой программы можно провести несколько экспериментов для исследования работы контроллера прерываний и системы прерываний в целом. К примеру, можно выполнить следующие операции:

- Изменить базовый адрес ведущего контроллера прерываний. Как мы обсуждали выше, BIOS инициализирует ведущий контроллер таким образом, что он имеет базовый адрес 08h. Попробуйте теперь изменить значение базового вектора, например, на значение 0f0h. Для этого необходимо выполнить инициализацию контроллера, которая заключается в последовательной посылке в него управляющих слов. Посмотрите последовательность приказов, которые BIOS посыпает в контроллер прерываний для его инициализации при загрузке системы (см. табл. 15.9). Нам тоже нужно будет их сформировать, но с нужными нам значениями, и послать в контроллер прерываний. Фрагмент, осуществляющий такие действия, может выглядеть следующим образом:

```
...
mov a1,00010001b
out 20h,a1 ;ICW1 в порт 20h
jmp $+2
jmp $+2      ;задержка, чтобы успела
               ;отработать аппаратура
mov a1,0f0h
out 21h,a1 ;ICW2 в порт 20h – новый базовый номер
jmp $+2
jmp $+2      ;задержка, чтобы успела
               ;отработать аппаратура
mov a1,00000100b
out 21h,a1 ;ICW3 – ведомый подключается
               ;к уровню 2 (см. рис. 15.1)
jmp $+2
jmp $+2      ;задержка, чтобы успела
               ;отработать аппаратура
```

```
mov a1,00000001b  
out 21h,a1      ;ICW4 – EOI выдает  
          ;программа пользователя
```

Данный фрагмент нужно вставить в начало процедуры `main` листинга 15.1 после команды `cli`. После этого вектору прерывания от таймера будет соответствовать значение `0f0h`. Соответственно, если вы хотите, чтобы программа листинга 15.1 работала как прежде, вам нужно настроить вектор `0f0h` на системную программу обработки прерывания `08h`. Техника такой замены аналогична приведенной в листинге 15.1. После этого можно разрешить прерывания командой `sti`. Но правильно работать будет только прерывание от таймера, все остальные прерывания (например, от клавиатуры) будут приводить к зависанию компьютера. Если вы подобным образом перепрограммировали контроллер, то перед завершением программы нужно провести обратное перепрограммирование, чтобы вернуть старое значение базового адреса. Если этого не сделать, то работа системы будет нарушена — все аппаратные прерывания будут попадать «не туда».

- Рассмотреть альтернативу команде `cli`, замаскировав аппаратные прерывания, используя прямое программирование регистра масок `IMR`:

```
;запретить прерывания  
mov a1,0ffh  
out 21h,a1      ;для ведущего контроллера  
out A1h,a1      ;для ведомого контроллера  
;разрешить прерывания  
mov a1,00h  
out 21h,a1      ;для ведущего контроллера  
out A1h,a1      ;для ведомого контроллера
```

Попробуйте использовать эти команды в листинге 15.1 вместо команд `cli` и `sti`.

- Запретить аппаратные прерывания определенных уровней. Например, в следующем фрагменте запрещаются прерывания от клавиатуры (см. табл. 15.6):

```
in   a1,21h  
or   a1,00000010b  
out  21h,a1
```

- Исследовать, как меняется содержимое регистров `IRR`, `IMR` и `ISR` в ходе обработки аппаратного прерывания, читая состояние описанных выше портов. Если у вас проснулся интерес к подобной исследовательской деятельности, то предлагаю вам самостоятельно написать эти фрагменты программ и исследовать их с использованием листинга 15.1.

Подведем некоторые итоги:

- Система прерываний микропроцессора Intel реализована весьма удачно. Ее применение позволяет достаточно гибко принимать и обрабатывать прерывания от различных источников.

- Источники прерываний делятся на внешние и внутренние. Количество внешних источников ограничено числом выводов микросхемы i8259A и не может превышать 15. К этому количеству нужно добавить еще одно прерывание — немаскируемое. Его инициируют источники, требующие безотлагательного вмешательства со стороны микропроцессора. Остальные источники прерываний являются внутренними. Общее количество источников прерываний в микропроцессоре не превышает 256. Внутренние источники прерываний также делятся на две группы: программные прерывания и исключения.
- Любое из этих прерываний можно вызвать как стандартными для этого вида прерывания средствами, так и командой `int xx`.
- Каждое прерывание связано с программой его обработки посредством таблицы векторов прерываний, которая в реальном режиме работы микропроцессора находится в первом килобайте оперативной памяти.
- Механизм обработки аппаратных прерываний основан на использовании микросхемы i8259A, которая позволяет организовать гибкую обработку прерываний.
- Микросхема i8259A является программируемой, что позволяет выполнить такие операции, как задание различных дисциплин обслуживания прерываний, запрещения отдельных прерываний и т. п.
- Программирование микросхемы i8259A осуществляется специальными последовательностями управляющих и операционных слов.

16

УРОК

Защищенный режим работы микропроцессора

- Характеристика защищенного режима работы микропроцессоров Intel
- Системные регистры микропроцессора
- Сегментные регистры и структуры данных защищенного режима
- Организация памяти в защищенном режиме
- Перевод микропроцессора в защищенный режим работы
- Основы программирования микропроцессора в защищенном режиме

На данном уроке мы рассмотрим *зашитенный* режим работы микропроцессора. Впервые защищенный режим появился в микропроцессоре i80286 фирмы Intel. Именно этот режим позволяет полностью использовать все возможности, предоставляемые микропроцессором. Все современные многозадачные операционные системы работают только в этом режиме. Такие операционные системы сейчас стали стандартом. Поэтому так важно для понимания всех процессов, происходящих в компьютере во время работы многозадачной операционной системы, знать основы функционирования микропроцессора в защищенном режиме.

Любой современный микропроцессор, находясь в реальном режиме, очень мало отличается от старого доброго i8086. Это лишь его более быстрый аналог с увеличенным (до 32 бит) размером всех регистров, кроме сегментных. Чтобы получить доступ ко всем остальным архитектурным и функциональным новшествам микропроцессора, необходимо перейти в защищенный режим. Если бы мы могли проникнуть внутрь компьютера после перехода в защищенный режим, то увидели бы, что микропроцессор совершенно преобразился. Прежде всего, это стало бы заметно в изменении принципов работы микропроцессора с памятью. Она по-прежнему является сегментированной, но изменяются функции и номенклатура программно-аппаратных компонентов, участвующих в сегментации. Вспомните, что в реальном режиме работы микропроцессора сегмент был длиной не более 64 Кбайт, а адрес области памяти сегмента располагался в одном из сегментных регистров. Функциональное назначение сегмента определялось тем, в каком из шести сегментных регистров находился его адрес. Аппаратные средства контроля доступа к сегменту отсутствовали. Если и можно было организовать такой контроль, то только со стороны операционной системы. Реальный режим поддерживал выполнение всего одной программы. Для этого достаточно было простых механизмов распределения оперативной памяти и не было потребности в организации защиты программ от взаимного влияния и т. д. Поэтому все, что нужно было знать программе, — это адреса, по которым располагаются ее сегменты кода, данных и стека. Если же мы вдруг захотели бы поместить в одну программно-аппаратную среду несколько независимых программ, то автоматически встал бы вопрос об их защите от взаимного влияния. Для решения этой проблемы микропроцессору уже недостаточно, используя сегментные регистры, знать, где располагаются сегменты программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, а если возникает потребность во взаимодействии между ними, то оно должно обязательно регулироваться.

Чтобы ввести такое регулирование, нужно иметь больше информации о самих задачах. Можно предложить несколько вариантов структурной организации и размещения такой информации. Фирма Intel не стала нарушать принцип сегмен-

тации. Так как каждая задача в системе занимает один или несколько сегментов в памяти, то логично иметь больше информации о них, как об объектах, реально существующих в данный момент в системе. Если каждому из этих объектов присвоить определенные атрибуты, то часть контроля за доступом к ним можно переложить на сам микропроцессор. Что и было сделано. Любой сегмент памяти в защищенном режиме имеет следующие атрибуты:

- расположение сегмента в памяти;
- размер сегмента;
- уровень привилегий — определяет права данного сегмента относительно других сегментов;
- тип доступа — определяет назначение сегмента;
- некоторые другие.

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

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

Это — ключевые моменты. Далее мы подробно рассмотрим архитектуру микропроцессора в защищенном режиме и основные правила программирования этого режима.

На уроке 2 было отмечено, что программная модель микропроцессора содержит 32 доступных пользователю регистра, которые делятся на две большие группы: пользовательские и системные. К настоящему моменту вы уже достаточно хорошо освоили работу с пользовательскими регистрами. Настало время разобраться с системными.

Системные регистры микропроцессора

Само название этих регистров говорит о том, что они выполняют специфические функции в системе. Использование этих регистров жестко регламентировано. Именно они обеспечивают работу защищенного режима. Их также можно рассматривать как часть архитектуры микропроцессора, которая намеренно оставлена видимой для того, чтобы квалифицированный системный программист мог выполнить самые низкоуровневые операции.

Системные регистры можно разделить на три группы:

- четыре регистра управления;
- четыре регистра системных адресов;
- восемь регистров отладки.

В состав системных регистров микропроцессоров ряда Pentium введены следующие изменения:

- задействован ранее зарезервированный регистр управления **cr4**;
- введена группа MSR-регистров (MSR – Model Specific Register, модельно-зависимые регистры процессора), назначение и возможности которых привязаны к архитектуре конкретной модели микропроцессора. Ранее их функции частично выполняли тестовые регистры, вошедшие теперь в состав MSR-регистров. Для доступа к этим регистрам введены специальные команды.

Регистры управления

В группу регистров управления входят пять регистров: **cr0**, **cr1**, **cr2**, **cr3**, **cr4**. Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0. Хотя микропроцессор имеет пять регистров управления, доступными являются только четыре из них; исключается **cr1**, функции которого пока не определены (он зарезервирован для будущего использования).

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

- pe** (Protect Enable), бит 0 – разрешение защищенного режима работы. Состояние этого флага показывает, в каком из двух режимов – реальном (**pe = 0**) или защищенном (**pe = 1**) – работает микропроцессор в данный момент времени. Запомните этот флаг, мы к нему еще вернемся;
- mp** (Math Present), бит 1 – наличие сопроцессора. Всегда 1;
- ts** (Task Switched), бит 3 – переключение задач. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи;
- am** (Alignment Mask), бит 18 – маска выравнивания. Этот бит разрешает (**am = 1**) или запрещает (**am = 0**) контроль выравнивания;
- cd** (Cache Disable), бит 30 – запрещение кэш-памяти. С помощью этого бита можно запретить (**cd = 1**) или разрешить (**cd = 0**) использование внутренней кэш-памяти (кэш-памяти первого уровня);
- pg** (PaGing), бит 31 – разрешение (**pg = 1**) или запрещение (**pg = 0**) страничного преобразования. Регистр **cr0** используется при страничной модели организации памяти.

Регистр **cr2** используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти. В такой ситуации в микропроцессоре возникает исключительная ситуация с но-

мером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр **cr2**. Имея эту информацию, обработчик исключения 14 определяет нужную страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы.

Регистр **cr3** также используется при страничной организации памяти. Это так называемый *регистр каталога страниц* первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь, каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайт.

Регистр **cr4** содержит признаки, в основном разрешительного характера, которые характеризуют те или иные архитектурные элементы, впервые появившиеся в различных моделях микропроцессоров Pentium. В качестве примеров таких свойств можно привести следующие: поддержка 36-разрядной адресации, использование отложенных прерываний в режиме виртуального i8086, поддержка страниц размером 4 Мбайт и т. д. Устанавливая в регистре **cr4** те или иные биты, можно включать или отключать поддержку этих свойств.

Регистры системных адресов

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

- **глобальное** — общее для всех задач;
- **локальное** — отдельное для каждой задачи.

Этим разделением и объясняется то, что в архитектуре микропроцессора присутствуют следующие системные регистры:

- регистр таблицы глобальных дескрипторов **gdtr** (Global Descriptor Table Register) имеет размер 48 бит и содержит 32-битный (биты 16–47) базовый адрес *глобальной дескрипторной таблицы* GDT и 16-битное (биты 0–15) значение *предела*, представляющее собой размер в байтах таблицы GDT;
- регистр таблицы локальных дескрипторов **ldtr** (Local Descriptor Table Register) имеет размер 16 бит и содержит так называемый *селектор дескриптора локальной дескрипторной таблицы* LDT. Этот селектор является указателем в таблице GDT, который описывает сегмент, содержащий *локальную дескрипторную таблицу* LDT.
- регистр таблицы дескрипторов прерываний **idtr** (Interrupt Descriptor Table Register) имеет размер 48 бит и содержит 32-битный (биты 16–47) базовый адрес *дескрипторной таблицы прерываний* IDT и 16-битное (биты 0–15) значение *предела*, представляющее собой размер в байтах таблицы IDT;
- 16-битный регистр задачи **tr** (Task Register) подобно регистру **ldtr** содержит *селектор*, то есть указатель на дескриптор в таблице GDT. Этот дескриптор

описывает текущий *сегмент состояния задачи* (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит *контекст* (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.

Регистры отладки

Это очень интересная группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486. Аппаратно микропроцессор содержит восемь регистров отладки, но реально из них используются только шесть.

Регистры dr0, dr1, dr2, dr3 имеют разрядность 32 бита и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах dr0...dr3, и при совпадении генерируется исключение отладки с номером 1.

Регистр dr6 называется регистром *состояния отладки*. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1. Перечислим эти биты и их назначение:

- b0 — если этот бит установлен в 1, то последнее исключение (прерывание) возникло в результате достижения контрольной точки, определенной в регистре dr0;
- b1 — аналогично b0, но для контрольной точки в регистре dr1;
- b2 — аналогично b0, но для контрольной точки в регистре dr2;
- b3 — аналогично b0, но для контрольной точки в регистре dr3;
- bd (бит 13) служит для защиты регистров отладки;
- bs (бит 14) устанавливается в 1, если исключение 1 было вызвано состоянием флага tf = 1 в регистре eflags;
- bt (бит 15) устанавливается в 1, если исключение 1 было вызвано переключением на задачу с установленным битом ловушки в TSS t = 1.

Все остальные биты в этом регистре заполняются нулями. Обработчик исключения 1 по содержимому dr6 должен определить причину, по которой произошло исключение, и выполнить необходимые действия.

Регистр dr7 называется регистром *управления отладкой*. В нем для каждого из четырех регистров контрольных точек отладки имеются поля, с помощью которых можно уточнить следующие условия, при которых следует сгенерировать прерывание:

- место регистрации контрольной точки — только в текущей задаче или в любой задаче. Соответствующие биты занимают младшие восемь бит регистра dr7 (по два бита на каждую контрольную точку (фактически, точку прерывания), задаваемую регистрами dr0, dr1, dr2, dr3). Первый бит из каждой пары — это так называемое *локальное разрешение*, его установка говорит о том, что точка прерывания действует, если она находится в пределах адресного пространства

текущей задачи. Второй бит в каждой паре определяет глобальное разрешение, которое говорит о том, что данная контрольная точка действует в пределах адресных пространств всех задач, находящихся в системе;

- тип доступа, по которому инициируется прерывание: только при выборке команды, при записи или при записи/чтении данных. Биты, определяющие природу возникновения прерывания, локализуются в старшей части данного регистра.

Большинство из системных регистров программно доступны. Не все из них понадобятся в нашем дальнейшем изложении, но, тем не менее, мы коротко рассмотрим их с тем, чтобы возбудить у читателя интерес к дальнейшему исследованию архитектуры микропроцессора.

Структуры данных защищенного режима

Мы уже упоминали о том, что в защищенным режиме любой запрос к памяти как со стороны операционной системы, так и со стороны прикладных программ должен быть санкционирован. Микропроцессор аппаратно контролирует доступ программ к любому адресу в оперативной памяти. Для получения доступа целевой адрес, к которому хочет получить доступ программа, должен быть описан для программы. Это означает, что участок физической памяти, содержащий нужный адрес, должен быть описан с помощью некоторого *дескриптора сегмента*, который помещается в одну из трех дескрипторных таблиц. Локализация этих таблиц осуществляется с использованием одного из рассмотренных нами системных регистров — `gdtr`, `ldtr` или `idtr`. Программе, которая желает использовать данный участок памяти, должен быть сообщен указатель на соответствующий дескриптор в одной из двух дескрипторных таблиц — GDT или LDT. Что касается таблицы IDT, то работа с ней осуществляется по несколько иному принципу, поэтому о ней мы поговорим позже. Указатель на дескриптор сегмента в одной из таблиц GDT или LDT, в зависимости от функционального назначения описываемого дескриптором участка памяти (сегмента), помещается в один из шести сегментных регистров. Таким образом, в защищенным режиме меняется роль сегментного регистра — теперь он содержит уже не адрес, а селектор или *индекс* в таблице дескрипторов сегментов. Но само назначение сегментных регистров не меняется — они по-прежнему указывают на сегменты команд, данных и стека, но делают это, используя принципиально иные механизмы.

Структурно дескриптор сегмента представляет собой 8-байтовую структуру, изображенную на рис. 16.1.

Поля в дескрипторе описаны в табл. 16.1.

Из табл. 16.1 видно, что в защищенным режиме размер сегмента не фиксирован, его расположение можно задать в пределах 4 Гбайт. Если посмотреть на рис. 16.1, то возникнет вопрос: почему разорваны поля, определяющие размер сегмента и его начальный (базовый) адрес? Это результат эволюции микропроцессоров. Мы упоминали о том, что защищенный режим впервые появился в микропроцессоре i80286. Этот микропроцессор имел 24-разрядную адресную шину и, соответственно, мог адресовать в защищенным режиме до 16 Мбайт оперативной памяти. Для

этого ему достаточно было иметь в дескрипторе поле базового адреса 24 бита и поле размера сегмента 16 битов. После появления микропроцессора i80386 с 32-разрядной шиной команд и данных в целях совместимости программ разработчики не стали изменять формат дескриптора, а просто использовали свободные поля. Внутри микропроцессора эти поля объединены. Внешние же они остались разделены, и при программировании с этим приходится мириться.

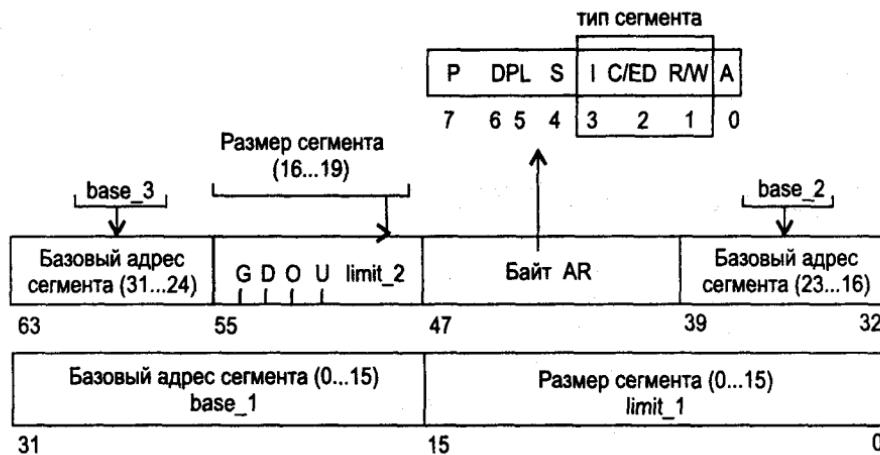


Рис. 16.1. Структура дескриптора сегмента защищенного режима микропроцессора

Следующий интересный момент связан с тем, что размер сегмента в защищенным режиме может достигать 4 Гбайт, то есть занимать все возможное физическое пространство памяти. Как это возможно, если суммарный размер поля размера сегмента всего 20 бит, что соответствует величине 1 Мбайт? Секрет скрыт в поле **гранулярности** – бит G (см. рис. 16.1 и табл. 16.1). Если бит G = 0, то значение в поле размера сегмента означает размер сегмента в байтах, а если G = 1, то в страницах. Размер страницы составляет 4 Кбайт. Нетрудно подсчитать, что когда максимальное значение поля размера сегмента составляет 0fffffh, то это соответствует 1 М страниц, что и соответствует величине 1 М * 4 Кбайт = 4 Гбайт.

Из высказанного понятно, что выведение информации о базовом адресе сегмента и его размере на уровень микропроцессора позволяет аппаратно контролировать работу программ с памятью и предотвращать обращения по несуществующим адресам либо по адресам, находящимся вне предела, разрешенного полем размера сегмента limit.

Другой аспект защиты заключается в том, что сегменты неравноправны в правах доступа к ним. Информация об этом содержится в специальном байте AR, входящем в состав дескриптора. Формат этого байта приведен в виде выноски на рис. 16.1. В табл. 16.2 дана краткая характеристика его полей.

Наиболее важные поля байта AR – это dpl и биты R/W, C/ED и I, которые вместе определяют тип сегмента. Поле dpl – часть механизма защиты по привилегиям. Суть этого механизма заключается в том, что конкретный сегмент может находиться на одном из трех уровней привилегированности с номерами 0, 1, 2 и 3.

Самым привилегированным является уровень 0. Существует ряд ограничений на взаимодействие сегментов с различными уровнями привилегий. В рамках данной книги мы считаем, что все сегменты находятся на нулевом уровне привилегий.

Таблица 16.1. Назначение полей дескриптора сегмента

| Номер байта в дескрипторе | Количество битов в поле | Символическое обозначение | Назначение и содержание полей дескриптора |
|---------------------------|-------------------------|---------------------------|---|
| 0...1 | 16 | limit_1 | Младшие биты 0...15 20-разрядного поля предела сегмента, который определяет размер сегмента в единицах, определяемых битом гранулярности g |
| 2...4 | 23 | base_1 | Биты 0...23 32-разрядной базы сегмента, которая определяет значение линейного адреса начала сегмента в памяти |
| 5 | 8 | AR | Байт, поля которого определяют права доступа к сегменту (табл. 16.2) |
| 6 | 0...3 | limit_2 | Старшие биты 16...19 20-разрядного предела сегмента |
| 6 | 1 | U | Бит пользователя (User). Не имеет специального назначения и может использоваться по усмотрению программиста |
| 6 | 1 | — | 0 — бит не используется |
| 6 | 1 | D | Бит разрядности операндов и адресов: 0 — в программе используются 16-разрядные операнды и режимы 16-разрядной адресации; 1 — в программе используются 32-разрядные операнды и режимы 32-разрядной адресации |
| 6 | 1 | G | Бит гранулярности: 0 — размер сегмента равен значению в поле limit в байтах; 1 — размер сегмента равен значению в поле limit в страницах |
| 7 | 8 | base_2 | Биты 24...31 32-разрядной базы сегмента |

Таблица 16.2. Байт AR дескриптора сегмента

| Номер бита в байте AR | Символическое обозначение | Назначение и содержимое |
|-----------------------|---------------------------|---|
| 0 | A | Бит доступа (Accessed) к сегменту. Устанавливается аппаратно при обращении к сегменту |
| 1 | R | Для сегментов кода — это бит доступа по чтению (Readable); определяет, возможно ли чтение из сегмента кода при осуществлении замены префикса сегмента: 0 — чтение из сегмента запрещено; 1 — чтение из сегмента разрешено |

| Номер бита в байте AR | Символическое обозначение | Назначение и содержимое |
|------------------------------|----------------------------------|---|
| | W | Для сегментов данных — это бит записи: 0 — модификация данных в сегменте запрещена; 1 — модификация данных в сегменте разрешена |
| 2 | C | Для сегментов кода — это бит подчинения (Conforming): 1 — подчиненный сегмент кода; 0 — обычный сегмент кода |
| | ED | Для многозадачного режима определяет особенности смены значения текущего уровня привилегий. Для сегментов данных — это бит расширения вниз (Expand Down); служит для различия сегментов стека и данных, а также определяет трактовку поля limit: 0 — сегмент данных; 1 — сегмент стека |
| 3 | I | Бит предназначения (Intending): 0 — сегмент данных или стека; 1 — сегмент кода |
| 4 | S | Если S = 1 — то это бит сегмента (Segment). Для любых сегментов в памяти равен 1. Назначение и порядок использования сегмента уточняется битами C и R; если = 0 — то это бит системный (System). Такое состояние бита S говорит о том, что данный дескриптор описывает специальный системный объект, который может и не быть сегментом в памяти |
| 5...6 | dpl | Поле уровня привилегий сегмента (Descriptor Privilege Level). Содержит численное значение в диапазоне от 0 до 3 привилегированности сегмента, описываемого данным дескриптором |
| 7 | P | Бит присутствия (Present): 0 — сегмента нет в оперативной памяти в данный момент; 1 — сегмент находится в оперативной памяти в данный момент |

Полю типа сегмента мы уделим больше внимания, так как оно понадобится нам при разработке программы. Это поле определяет целевое назначение сегмента. В табл. 16.2 биты, входящие в состав поля типа, разделены, но для облегчения их совместной интерпретации обычно рассматривают комбинации этих битов в целом. В табл. 16.3 перечислено назначение некоторых комбинаций этих битов.

Как видно из содержимого поля типа в байте AR, возможны два принципиально разных вида сегментов: данных и кода. Сегмент стека является разновидностью сегмента данных, но с особой трактовкой поля размера сегмента. Это объясняется спецификой использования стека (он растет в направлении младших адресов памяти). Таким образом, видно, что поле типа ограничивает использование объявленных сегментов. В частности, программные сегменты не могут быть модифи-

цированы без применения специальных приемов. Доступ к сегменту данных также может быть ограничен только на чтение.

Таблица 16.3. Типичные комбинации в поле типа сегмента

| Комбинации битов в поле type_seg | Назначение сегмента |
|-------------------------------------|--|
| 000 | Сегмент данных, только для чтения |
| 001 | Сегмент данных с разрешением чтения и записи |
| 010 | Не определено |
| 011 | Сегмент стека с разрешением чтения и записи |
| 100 | Сегмент кода с разрешением только выполнения |
| 101 | Сегмент кода с разрешением выполнения и чтения из него |
| 110 | Подчиненный сегмент кода с разрешением выполнения |
| 111 | Подчиненный сегмент кода с разрешением выполнения и чтения из него |

Итак, мы выяснили, что в защищенном режиме перед использованием любой области памяти должна быть проведена определенная работа по инициализации соответствующего дескриптора. Эту работу выполняет операционная система или программа, сегменты которой также описываются подобными дескрипторами.

Интерес представляет то, каким образом микропроцессор переходит в защищенный режим. Сразу после включения питания или нажатия кнопки сброса микропроцессор начинает свою работу в реальном режиме. В этом режиме он производит действия по тестированию аппаратуры компьютера. После успешного завершения тестирования микропроцессор выполняет начальную загрузку системы, используя программу начальной загрузки, хранящейся на нулевой дорожке диска. Программа начальной загрузки считывает с диска программу инициализации операционной системы и передает ей управление. Действие этой программы зависит от того, в каком режиме работы микропроцессора будет осуществляться дальнейшее функционирование системы. Если в реальном режиме, то операционная система формирует среду и структуры данных для работы в этом режиме. Если же загружаемая операционная система собирается дальше работать в защищенном режиме, то она должна в него специальным образом перейти. Но прежде чем сделать это, операционная система формирует системные структуры данных (в частности, рассмотренные нами дескрипторные таблицы) для работы в защищенном режиме. Затем можно переходить в защищенный режим и выполнять дальнейшие действия. На этом уроке мы с вами попытаемся написать некоторый фрагмент такой операционной системы, а именно тот, который производит инициализацию структур защищенного режима и перевод микропроцессора в этот режим. Если у вас возникнет желание, то далее вы можете написать свою маленькую операционную систему, выполняющую некоторые действия.

Вначале — несколько организационных моментов. Понятно, что в своих действиях мы должны следовать логике, в соответствии с которой микропроцессор переводится в защищенный режим. Исходя из нее, наша операционная мини-система

должна начинать свою работу в реальном режиме, поэтому и наша программа должна начать работать в среде такой операционной системы. Особого выбора у нас нет: операционная система MS-DOS либо режим эмуляции MS-DOS в Windows 95. Запустить программу прямо из многозадачной операционной системы, которой является, например, Windows 95, нельзя. Дело в том, что в целях защиты любая операционная система защищенного режима перекрывает на определенном уровне доступ к системным ресурсам со стороны программ пользователя с тем, чтобы не нарушить свою работу. Делается это следующим образом. Мы с вами упоминали об уровне привилегий, как одном из атрибутов сегментов памяти в защищенном режиме. В этом режиме наша программа, написанная как приложение MS-DOS, будет выполняться в режиме виртуального 8086 с уровнем привилегий 3. Чтобы получить доступ к ресурсам микропроцессора, управляющим сегментацией, ей необходимо иметь уровень привилегий 0. Кроме того, находясь в защищенном режиме, не имеет смысла говорить о проблеме перехода в него.

Поэтому у нас есть две возможности — загрузить MS-DOS с помощью системной дискеты либо выполнить перезагрузку своего компьютера с тем, чтобы перевести Windows 95 в режим эмуляции MS-DOS.

Но одной загрузки MS-DOS мало, необходимо проследить, чтобы в файле config.sys были отключены все драйверы расширенной памяти типа emm386.exe, так как они неявно переводят микропроцессор в защищенный режим работы, исключая доступ к структурам данных этого режима. Впрочем, если вы забудете это сделать, то при первой же попытке вашей программы перейти в защищенный режим вы получите соответствующее сообщение. Закомментируйте соответствующие строки в файле config.sys, и проблема будет снята.

Пример программы защищенного режима

Во второй половине урока мы рассмотрим фрагменты нашей операционной мини-системы, которые подготавливают структуры данных защищенного режима, переводят микропроцессор в этот режим, имитируют работу путем вывода некоторого сообщения, после чего переводят микропроцессор обратно в реальный режим и заканчивают выполнение. Давайте перечислим и обсудим действия, необходимые для обеспечения функционирования этой программы:

1. Подготовка в оперативной памяти таблицы глобальных дескрипторов GDT.
2. Инициализация необходимых дескрипторов в таблице GDT.
3. Загрузка в регистр gdtr адреса и размера таблицы GDT.
4. Запрет обработки аппаратных прерываний.
5. Переключение микропроцессора в защищенный режим.
6. Организация работы в защищенном режиме:
 - настроить сегментные регистры;
 - выполнить собственно содержательную работу программы; в нашем случае мы просто обозначим сам факт успешного перехода в защищенный режим;

- под готовиться к возврату в реальный режим;
 - запретить аппаратные прерывания.
7. Переключение микропроцессора в реальный режим.
8. Настройка сегментных регистров для работы в реальном режиме.
9. Разрешение прерываний.
10. Стандартное для MS-DOS завершение работы программы.

Далее мы подробно обсудим каждый этап и суммируем все в виде листинга (см. текст программы `prg16_1.asm` на диске в каталоге данного урока).

Подготовка таблиц глобальных дескрипторов GDT

Для того чтобы собрать информацию о всех программных объектах, находящихся в данный момент в памяти и в системе в целом, их дескрипторы собираются в таблицы, которые представляют собой массивы 8-байтовых элементов.

Микропроцессор аппаратно поддерживает три типа дескрипторных таблиц:

1. Таблица GDT (Global Descriptor Table) — глобальная дескрипторная таблица. Это основная общесистемная таблица, к которой допускается обращение со стороны программ, обладающих достаточными привилегиями. Расположение таблицы GDT в памяти произвольно; оно локализуется с помощью специального регистра `gdtr`. В таблице GDT могут содержаться следующие типы дескрипторов:
 - дескрипторы сегментов кодов программ;
 - дескрипторы сегментов данных программ;
 - дескрипторы стековых сегментов программ;
 - дескрипторы TSS (Task Segment Status) — специальные системные объекты, называемые сегментами состояния задач;
 - дескрипторы для таблиц LDT;
 - шлюзы вызова;
 - шлюзы задач.
2. Таблица LDT (Local Descriptor Table) — локальная дескрипторная таблица. Для любой задачи в системе может быть создана своя дескрипторная таблица, подобно общесистемной GDT. Тем самым адресное пространство задачи локализуется в пределах, установленных набором дескрипторов таблицы LDT. Для связи между таблицами GDT и LDT в таблице GDT создается дескриптор, описывающий область памяти, в которой находится LDT. Расположение таблицы LDT в памяти также произвольно и локализуется с помощью специального регистра `ldtr`. В таблице LDT могут содержаться следующие типы дескрипторов:
 - дескрипторы сегментов кодов программ;
 - дескрипторы сегментов данных программ;
 - дескрипторы стековых сегментов программ;

- шлюзы вызова;
 - шлюзы задач.
3. Таблица IDT (Interrupt Descriptor Table) – дескрипторная таблица прерываний. Данная таблица также является общесистемной и содержит дескрипторы специального типа, которые определяют местоположение программ обработчиков всех видов прерываний. В качестве аналогии можно привести таблицу векторов прерываний реального режима. Расположение таблицы IDT в памяти произвольно и локализуется с помощью специального регистра `idtr`. Элементы данной таблицы называются *шлюзами*. Это элементы особого рода, и мы их рассмотрим на следующем уроке при изучении обработки прерываний в защищенном режиме. Сейчас только отметим, что эти шлюзы бывают трех типов:
- шлюзы задач;
 - шлюзы прерываний;
 - шлюзы ловушек.

Каждая из дескрипторных таблиц может содержать до 8192 (2^{13}) дескрипторов. Данное значение определяется размерностью поля в сегментном регистре. Мы помним, что в защищенном режиме роль дескрипторов изменяется по сравнению с реальным режимом, и это отражается даже на их названии – в защищенном режиме они называются селекторами сегментов.

Так как наша программа претендует лишь на роль фрагмента операционной системы, то вполне достаточно определить пока только одну дескрипторную таблицу – глобальную дескрипторную таблицу GDT. Прерывания мы пока обрабатывать не будем – подождем до следующего урока. Таблицу LDT есть смысл применять, когда в системе работают несколько задач и необходимо изолировать их друг от друга. Пока от нас этого не требуется. Тем не менее отметим следующее.

Выше мы сказали, что дескрипторы, описывающие сегменты некоторой программы, могут содержаться как в глобальной (GDT), так и в локальной (LDT) дескрипторных таблицах. Сегментные регистры содержат селекторы, которые являются указателями на дескрипторы, описывающие соответствующие области памяти. Но как микропроцессор узнает о том, в какой из двух дескрипторных таблиц находится дескриптор, на который указывает селектор?

Структура сегментного регистра в защищенном режиме представляется тремя полями. Поле RPL, занимающее два младших бита 0 и 1 (Request Privilege Level – запрашиваемый уровень привилегий), используется в механизме ограничения доступа по привилегиям. А вот состояние однобитового поля T I, занимающего бит 2 сегментного регистра, как раз и определяет, с какой именно таблицей идет работа:

- если $T I = 0$, то сегментный регистр содержит селектор на дескриптор в глобальной дескрипторной таблице GDT;
- если $T I = 1$, то сегментный регистр содержит селектор на дескриптор в локальной дескрипторной таблице LDT.

И наконец, сегментный регистр содержит поле селектора. Оно определяет число, кратное восьми (так как три младшие бита заняты под поля RPL и TI), являющееся указателем на дескриптор в одной из дескрипторных таблиц в соответствии со значением бита в поле TI. Управлять состоянием этого бита может либо сама программа, но тогда она должна обладать достаточным уровнем привилегий, либо операционная система, обеспечивающая ее работу. Для самой же программы ничего не меняется. По-прежнему базовые адреса ее сегментов определяются с помощью сегментных регистров, хотя и по-иному, чем в реальном режиме, принципу. В каждый момент времени микропроцессор может работать только с одной дескрипторной таблицей: GDT или LDT. В нашем случае мы будем считать, что все сегменты программы находятся в глобальной дескрипторной таблице GDT, то есть TI = 0 во всех используемых сегментных регистрах.

Определимся теперь с набором дескрипторов в таблице GDT, которые понадобятся для нашей программы:

- дескриптор для описания сегмента самой таблицы GDT;
- дескриптор для описания сегмента данных программы;
- дескриптор для описания сегмента команд программы;
- дескриптор для описания сегмента стека программы;
- дескриптор для описания сегмента, в котором будет находиться процедура для выдачи сигнала сирены;
- дескриптор для описания видеопамяти.

Формат дескриптора сегмента показан на рис. 16.1. В программе его удобнее описать в виде структуры:

```
descr      struc
limit      dw      0
base_1     dw      0
base_2     db      0
atrdb      0
lim_atr   db      0
base_3     db      0
ends
```

Используя этот шаблон структуры, опишем таблицу GDT как массив структур:

```
gdt_seg    segment    para
gdt_0      descr <0,0,0,0,0,0>
gdt_gdt_8 descr <0,0,0,0,0,0>
gdt_ldt_10 descr <0,0,0,0,0,0>
gdt_ds_18 descr <0,0,0,0,0,0>
gdt_es_vbf_20 descr <0,0,0,0,0,0>
gdt_ss_28 descr <0,0,0,0,0,0>
gdt_cs_30 descr <0,0,0,0,0,0>
gdt_size=$-gdt_0-1 :определение размера таблицы GDT
gdt_seg ends
```

Инициализация дескрипторов в таблице GDT

После того как мы определились с набором дескрипторов и даже сформировали глобальную дескрипторную таблицу, нужно разобраться с тем, где брать информацию для их заполнения и как это делать. Давайте обсудим этот вопрос для каждого поля в отдельности:

- поле `limit` – размер сегмента. В это поле нужно поместить точный размер сегмента за вычетом единицы, так как адресация идет от нуля. Наиболее оптимальный способ заключается в использовании оператора `$` – извлечь текущее значение счетчика адреса. Пример использования этого оператора для определения размера приведен выше при описании таблицы GDT;
- поля `base_1`, `base_2`, `base_3` – поля 32-разрядного базового адреса. Этот адрес будет известен после загрузки программы в оперативную память, и поля придется загружать на этапе выполнения при подготовке к переходу в защищенный режим. Фрагмент программы заполнения поля базового адреса для дескриптора `gdt_gdt_8` может выглядеть так:

```
xor    eax, eax
mov    ax, gdt_seg ;адрес сегмента в ax
;сдвигом на 4 разряда получим физический
;20-разрядный адрес сегмента gdt_seg:
shl    eax, 4
mov    base_1, ax
rol    eax, 16 ;меняем для получения
               ;оставшейся части адреса
mov    base_2, al
```

Так как эту операцию нам придется проделывать несколько раз, для каждого из сегментов программы, то для удобства работы и повышения наглядности оформим этот фрагмент в виде макроса `load_addr`:

```
load_addr macro descr, seg_addr, seg_size
  mov    descr.limit, seg_size
  xor    eax, eax
  mov    ax, seg_addr
  shl    eax, 4
  mov    descr.base_1, ax
  rol    eax, 16
  mov    descr.base_2, al
  endm
```

Заодно, как видите, мы добавили к операциям, выполняемым макросом, и инициализацию поля предела;

- поле `atr` – байт атрибутов. Структура этого байта представлена в табл. 16.2. Понятно, что для конкретного сегмента его значение будет всегда константой. Чтобы не запутывать себя окончательно, на мой взгляд, значение этого байта в целом удобнее формировать как логическую сумму нужных значений его полей для определения типа конкретного сегмента. Сформируем элементарные константы полей этого байта в соответствии с битами табл. 16.2:

```
;биты 0, 4, 5, 6, 7 – постоянная часть
;bайта AR для всех типов сегментов
```

```

const equ 10010000b
;бит 1 – доступность сегментов по чтению/записи
code_r_n equ 00000000b ;сегмент кода:
;чтение запрещено
code_r_y equ 00000010b ;сегмент кода:
;чтение разрешено
data_wm_n equ 00000000b ;сегмент данных:
;модификация запрещена
data_wm_y equ 00000010b ;сегмент данных:
;модификация разрешена

;бит 2 – тип сегмента
code_nequ 00000000b ;обычный сегмент кода
code_pequ 00000100b ;подчиненный сегмент кода
_data equ 00000000b ;сегмент данных
_stack equ 00000100b ;сегмент стека
;бит 3 – предназначение
_code equ 00001000b ;сегмент кода
data_stk equ 00000000b ;сегмент данных или стека

```

Теперь для получения значения атрибута для нужного сегмента достаточно подобрать нужные константы по битам и выполнить подсчет суммы, как, например, для дескриптора `gdt_gdt_8`:

```
atr=const or data_wm_y or _data or data_stk
```

Значение `atr` для дескриптора сегмента `gdt_gdt_8` будет равно `10010010 = = 92h`. Для удобства использования можно создать макрос, который будет создавать константу с именем, состоящим из двух частей: приставки `atr` и имени дескриптора, для которого формируется байт атрибута (к примеру, для дескриптора `gdt_gdt_8` это будет `atr_gdt_gdt_8`):

```
atr macro descr,bit1,bit2,bit3
atr&descr=const or bit1 or bit2 or bit3
endm
```

Имена формируемых констант нужно указывать при инициализации структур для каждого дескриптора, к примеру, для `gdt_gdt_8`:

```

gdt_seg segment para
gdt_0 descr <0,0,0,0,0,0>
    atr gdt_gdt_8,data_wm_y,_data,data_stk
gdt_gdt_8 descr <0,0,0,atr_gdt_gdt_8,0,0>
;...
gdt_seg ends

```

- поле `lim_atr` – байт, состоящий из четырех старших битов размера сегмента и четырех атрибутов (см. табл. 16.1). В нашем случае размер сегмента небольшой, то есть четыре старших бита размера равны 0. И оставшиеся биты атрибутов для нашего случая также нулевые;
- поле `base_3` содержит старший байт 4-байтового физического адреса сегмента. Так как мы начинаем работу в реальном режиме, где размер максимального физического адреса не превышает 20 бит, то этот байт также будет нулевым.

Таким образом, в нашей программе инициализации будут подлежать три поля: `limit`, три первых байта адреса `base_1` и `base_2` и байт атрибута `atr`.

Загрузка регистра gdtr

В начале урока мы обсуждали набор и назначение системных регистров. Также мы говорили о том, что дескрипторные таблицы являются ключевыми для организации работы в защищенном режиме. Их местоположение в памяти может быть любым. Микропроцессор узнает о том, где находятся эти таблицы, по содержимому определенных системных регистров. Так, после того как нами была сформирована таблица GDT, ее адрес нужно поместить в регистр `gdtr`. Но этого мало, так как в этот же регистр нужно поместить и размер этой таблицы. Для загрузки именно этого регистра в системе команд микропроцессора есть специальная команда:

`1gdt адрес_48-битного поля` (Load GDT register) — загрузить регистр `gdtr`.

Команда `1gdt` загружает системный регистр `gdtr` содержимым 6-байтового поля, адрес которого указан в качестве операнда.

Из описания команды следует, что вначале необходимо сформировать поле из шести байт со структурой, аналогичной формату регистра `gdtr`, а затем указать адрес этого поля в качестве операнда команды `1gdt`.

Для резервирования поля из шести байт (48 бит) TASM поддерживает специальные директивы резервирования и инициализации данных — `dri` и `df`. После выделения — с помощью одной из этих директив — области памяти в сегменте данных необходимо сформировать в этой области указатель на начало таблицы GDT и ее размер. Но удобнее использовать структуру. Пример ее использования показан в следующем фрагменте программы:

```
point      struc
l1mdw      0
adrdd      0
    ends
data        segment
point_gdt point <gdt_size,0>
:...
code        segment
:...
    xor     eax,eax
    mov     ax,gdt_seg
    shl     eax,4
    mov     dword ptr point_gdt.adr,eax
1gdt      pword point_gdt
```

Запрет обработки аппаратных прерываний

Как мы увидим на следующем уроке, обработка прерываний в защищенном режиме принципиально отличается от обработки прерываний в реальном режиме (см. урок 15). Поэтому, как только микропроцессор переключится в защищенный режим, первое же прерывание от таймера, которое происходит 18,2 раза в секунду, «подвесит» компьютер. До тех пор пока мы не научимся обрабатывать прерывания как программные, так и внешние, их нужно будет запрещать. Для этого вы уже освоили два способа: прямым программированием контроллера прерываний

и командой микропроцессора с1 i. Можете использовать любой, только не нужно их сочетать.

Переключение микропроцессора в защищенный режим

Теперь у нас все готово для того, чтобы корректно перейти в защищенный режим. Специальных команд микропроцессора для выполнения такого перехода нет. О том, что микропроцессор находится в защищенном режиме, говорит лишь состояние бита ре в регистре cr0. Установить этот бит можно двумя способами:

- непосредственной установкой бита ре в регистре cr0. Состояние этого бита управляет режимами работы микропроцессора: если ре = 0, то микропроцессор работает в реальном режиме, если ре = 1, то микропроцессор работает в защищенном режиме;

- использованием функции 89h прерывания 15h BIOS.

Мы опишем оба способа, но использовать будем первый.

Регистр cr0 программно доступен, поэтому установить бит ре можно, используя обычные команды ассемблера:

```
mov    eax,cr0  
or     eax,0001h  
mov    cr0,eax
```

Последняя команда mov переводит микропроцессор в защищенный режим.

Функция 89h прерывания 15h выполняет это и некоторые другие действия неявно. Прежде чем вызывать это прерывание, необходимо посредством регистров сообщить ему следующее:

- ah = 89h;
- b1 = новый номер для аппаратного прерывания уровня iqf0. Уровни iqf1...7 будут иметь следующие по порядку номера;
- bh = новый номер для аппаратного прерывания уровня iqf8. Уровни iqf9..f будут иметь следующие по порядку номера;
- ds:s1 = адрес GDT для защищенного режима;
- cx = адрес первой выполняемой команды в защищенном режиме.

Эта функция предполагает, что дескрипторы в таблице GDT расположены в определенной последовательности:

- 0h – пустой дескриптор;
- 8h – дескриптор таблицы GDT;
- 10h – дескриптор таблицы LDT;
- 18h – дескриптор сегмента данных, на него указывает селектор в регистре ds;
- 20h – дескриптор дополнительного сегмента данных, на него указывает селектор в регистре es;
- 28h – дескриптор сегмента стека, на него указывает селектор в регистре ss;

- 30h – дескриптор сегмента кода, на него указывает селектор в регистре с s;
- остальные дескрипторы.

В нашей таблице GDT этот порядок следования соблюден, поэтому фрагмент программы перевода микропроцессора в защищенный режим может быть следующим:

```
code segment
...
    mov    ah,89h
    mov    bl,20h
    mov    bh,28h
    mov    ax,gdt_seg
    mov    ds,ax
    mov    si,0
    lea    cx,protect
    int    15h
protect:
... ;работа в защищенном режиме
```

Работа в защищенном режиме

Настройка сегментных регистров

После выполнения всех вышеописанных действий состояние микропроцессора можно сравнить с состоянием человека, которого после пребывания на ярком солнце завели в темную комнату. Что нужно сделать в такой ситуации для того, чтобы микропроцессор не уподобился слону в посудной лавке?

Мы с вами уже разобрались, что содержимое сегментных регистров в реальном и защищенном режимах интерпретируется микропроцессором по-разному. Как только микропроцессор оказывается в защищенном режиме, первую же команду он пытается выполнить традиционно: по содержимому пары с s : iр определить ее адрес, выбрать ее и т. д. Но содержимое с s должно быть индексом, указывающим на дескриптор сегмента кода в таблице GDT. Но пока это не так, так как в данный момент с s все еще содержит физический адрес параграфа сегмента кода, как этого требуют правила формирования физического адреса в реальном режиме. То же самое происходит и с другими регистрами. Но если содержимое других сегментных регистров можно подкорректировать в программе, то в случае с регистром с s этого сделать нельзя, так как он в защищенном режиме программно недоступен. Нужно помочь микропроцессору сориентироваться в этой затруднительной ситуации. Вспомним, что действие команд перехода основано как раз на изменении содержимого регистров с s и iр. Команды ближнего перехода изменяют только содержимое e iр \ iр, а команды дальнего перехода – обоих регистров, с s и e iр \ iр. Воспользуемся этим обстоятельством, в добавок существует и еще одно свойство команд передачи управления – они сбрасывают конвейер микропроцессора, подготавливая его тем самым к приему команд, которые сформированы уже по правилам защищенного режима. Это же обстоятельство заставляет нас впрямую моделировать команду межсегментного перехода, чтобы поместить правильное значение селектора в сегментный регистр с s. Это можно сделать так:

```

code segment
...
db 0eah      ; машинный код команды jmp
dw offset protect ; смещение метки перехода
; в сегменте команд
dw 30h      ; селектор сегмента кода в таблице GDT
protect:
;загрузить селекторы для остальных дескрипторов
mov ax,18h
mov ds,ax      ;сегментный регистр данных
mov ax,28h
mov ss,ax      ;сегментный регистр стека
mov ax,20h
mov es,ax      ;дополнительный сегмент данных:
;для указания на видеобуфер

```

После этого можно работать так, как мы уже привыкли.

Но за кадром остался один момент, на который нужно обязательно обратить внимание. В микропроцессоре каждому сегментному регистру соответствует свой *теневой регистр дескриптора*. Этот регистр имеет размер 64 бита и формат дескриптора сегмента. Смена содержимого теневых регистров производится автоматически всякий раз при смене содержимого соответствующего сегментного регистра. Последние наши действия по изменению содержимого сегментных регистров привели к тому, что мы неявно записали в теневые регистры микропроцессора содержимое соответствующих дескрипторов из GDT. Программисту теневые регистры недоступны, с ними работает только микропроцессор. Если есть необходимость изменить их содержимое, то для этого нужно сначала изменить сам дескриптор, а затем загрузить соответствующий ему селектор в нужный сегментный регистр.

Выполнение собственно программы защищенного режима

Наша программа должна будет выполнить несколько действий. При этом есть существенные ограничения на ее работу. Прежде всего, это связано с тем, что пока мы не умеем обрабатывать прерывания. Так что придется использовать прямой доступ к аппаратуре, используя пространство портов ввода-вывода (см. текст программы prg16_1.asm на диске в каталоге данного урока).

Подготовка к возврату в реальный режим

Здесь возникает примерно та же проблема с сегментными регистрами, что была при входе в защищенный режим. Мы упоминали уже о теневых регистрах микропроцессора, но не сказали того, что микропроцессор использует их, даже работая в реальном режиме. При этом поля этих регистров заполнены, конечно, в соответствии с требованиями реального режима:

- предел должен быть равен 64 К = 0ffffh;
- бит G = 0 (значение размера в поле предела) — это значение в байтах;
- байт атрибута равен '10010010 = 92h;
- базовый адрес значения не имеет.

Следовательно, перед переходом в реальный режим нужно сформировать эти значения в соответствующих дескрипторах и сделать актуальными эти изменения в теневых регистрах, для чего нужно перезагрузить сегментные регистры. После этого можно смело переходить в реальный режим. В программе все эти действия могут выглядеть так:

```
<1>      code segment
<2>      ...
<3>      ;мы в защищенном режиме и начинаем переход в реальный режим
<4>      ;загрузим значение 0ffffh в поле предела
<5>      mov    gdt_ds_18.limit,0ffffh
<6>      ;для регистров cs, ss, es, fs и gs аналогично
<7>      ;селектор - в регистр данных, чтобы обновить теневой регистр для ds
<8>      mov    ax,18
<9>      mov    ds,ax
<10>     ;для регистров ss, es, fs и gs аналогично
<11>     ;для регистра cs загрузку селектора проведем
<12>     ;особым способом - командой jmp far
<13>     db    0eah
<14>     dw    offset jump
<15>     dw    30h
<16>     jump:
<17>     ;можно переходить в реальный режим
```

Запрет аппаратных прерываний

Для нашей программы этого делать и не нужно, так как мы их и не разрешали. Но в будущем вы должны иметь в виду, что перед переходом в реальный режим, кроме настройки сегментных регистров, нужно будет перестраивать и систему прерываний. Для этого на участке программы, производящем соответствующие действия, необходимо запретить аппаратные прерывания одним из известных вам способов.

Переключение микропроцессора в реальный режим

Скорее всего, что после нашего обсуждения выполнение этой операции не составит для вас труда. Нужно всего лишь сбросить нулевой бит регистра cr0. Это можно сделать несколькими способами. К примеру:

```
mov    eax,cr0
and    al,0feh
mov    cr0, eax
```

После сброса бита ре микропроцессор снова оказался в реальном режиме.

Настройка сегментных регистров

После перехода в реальный режим опять возникает проблема с содержимым сегментных регистров. Решается она уже известным вам способом:

```
;...
;моделирование команды дальнего перехода для загрузки cs
```

```
db    0eah  
dw    real_mode  
dw    code  
real_mode:  
    mov   ax,data  
    mov   ds,ax  
    mov   ax,stk  
    mov   ss,ax
```

Разрешение прерываний

Теперь можно разрешить прерывания. Так как в системе прерываний мы ничего не меняли, то действия наши тоже были простейшими: а) запрещение аппаратных прерываний для того, чтобы на время смены режима работы микропроцессора нас не беспокоило прерывание от таймера, б) последующее разрешение прерываний после возврата в реальный режим. Что может быть легче? На практике, конечно, без прерываний не обойтись, и, как мы увидим на следующем уроке, этот вопрос не менее сложен, чем перевод микропроцессора из одного режима работы в другой. Пока же все просто:

```
c1i ;флаг IF в 0 – запретить прерывания от аппаратуры
```

и

```
sti ;флаг IF в 1 – разрешить прерывания от аппаратуры
```

Стандартное для MS-DOS завершение работы программы

Этот вопрос комментировать не имеет смысла. Если все было сделано правильно, то окончание работы программы выглядит стандартно.

Теперь нам осталось собрать все вместе в одну программу, ввести ее в машину и приступить к экспериментам. Нужно отметить один момент, с которым вы столкнетесь при трансляции исходного модуля. Он связан с вычислением константы `code_size`. Фрагмент листинга с этой ошибкой будет выглядеть так:

```
184 load_descr gdt_cs_30,CODE,code_size  
1 185 0072 C706 0030r 0197 mov gdt_cs_30.limit,code_size  
**Error** prg_16_1.asm(94) LOAD_DESCR(1) Forward reference needs override
```

Природа этой ошибки понятна — опережающая ссылка на объект, о котором компилятор еще не имеет информации. Объектом в данном случае является константа `code_size`. Она вычисляется в конце сегмента кода, а применяется — в середине. Чтобы устранить эту ошибку, нужно заставить компилятор выполнить два прохода программы. Тогда за первый проход он получит представление обо всех объектах программы, а на втором проходе сформирует правильные команды. Чтобы указать компилятору на необходимость выполнения двух проходов, используется опция командной строки `/m2`:

```
tasm /m2 prg16_1.asm,,
```

Полный текст программы перевода микропроцессора в защищенный режим ([prg16_1.asm](#)) можно найти на диске в каталоге данного урока.

Подведем некоторые итоги:

- Микропроцессор в защищенном режиме раскрывает все свои возможности. В этом режиме он позволяет увеличить объем адресуемой оперативной памяти до 4 Гбайт, поддерживает несколько моделей организации памяти: плоскую, многосегментную и страничную.
- Основное достоинство защищенного режима — поддержка мультизадачности и еще одного, третьего, режима работы микропроцессора — *режима виртуального i8086*. Этот режим позволяет работать параллельно нескольким программам, разработанным для микропроцессора i8086.
- Многозадачность поддерживается микропроцессором на аппаратном уровне. Для этого он имеет специальные системные регистры. В некоторые из этих регистров должны быть загружены адреса системных таблиц.
- Системные таблицы состоят из дескрипторов, которые описывают все используемые в данный момент в системе области памяти.
- Каждый дескриптор представляет собой структуру, которая определяет адрес участка памяти, его размер и ряд атрибутов, регулирующих доступ к нему.
- После включения или сброса микропроцессор работает в реальном режиме. Для того чтобы микропроцессор начал функционировать в защищенном режиме, необходимо выполнить ряд действий. Их цель — формирование структур защищенного режима, настройка некоторых системных регистров. Лишь после этого микропроцессор можно переключить в защищенный режим.

17

УРОК

Обработка прерываний в защищенном режиме

- Перечень и свойства прерываний в защищенном режиме
- Схема обработки прерываний в защищенном режиме
- Формат таблицы дескрипторов IDT и типы дескрипторов
- Практические проблемы обработки прерываний в защищенном режиме

Этот урок мы посвятим рассмотрению того, как организована обработка прерываний в защищенном режиме работы микропроцессоров Intel. Это будет логическим завершением рассмотрения архитектуры микропроцессора (хотя мы не стали рассматривать средства поддержки многозадачности, защиты задач, аппаратной отладки, режим виртуального i8086 и многое другое, так как все это темы отдельного большого разговора).

Кому-то из читателей может показаться, что на последних трех уроках мы отклонились от рассмотрения основных вопросов книги — изучения непосредственно языка ассемблера, его конструкций, различных приемов и т. д. Сделали мы это намеренно, так как основная область применения языка ассемблера — разработка различных системных программ. А как известно, лучший способ обучения — это хорошие практические примеры. Поэтому мы намеренно перевели наше изложение в чисто практическую плоскость, где читатели могут самостоятельно попробовать свои силы, выработать свой стиль и т. д. Ведь вы уже поняли, что одно и то же действие на ассемблере можно реализовать по крайней мере несколькими способами.

На уроке 15 мы обсудили понятие прерывания в микропроцессоре и то, как организуется обработка прерываний в реальном режиме. Обработка прерываний в защищенном режиме отличается от обработки в реальном режиме так же сильно, как и сам защищенный режим отличается от реального. Причины здесь в том, что, во-первых, в защищенном режиме несколько иное распределение номеров векторов прерываний и, во-вторых, здесь принципиально иным является сам механизм обработки прерываний.

Для того чтобы разобраться с первой причиной, посмотрим на распределение векторов прерываний в защищенном режиме. Чтобы сориентироваться в терминологии, посмотрите классификацию прерываний в уроке 15. В табл. 17.1 для удобства сравнительного анализа приведено распределение номеров прерываний в реальном и защищенном режимах.

Таблица 17.1. Распределение прерываний в защищенном и реальном режимах

| Номер вектора | Реальный режим | Защищенный режим | Тип прерывания | Код ошибки |
|---------------|------------------|------------------|--------------------|----------------|
| 0 | Деление на ноль | Деление на ноль | Ошибка | Не формируется |
| 1 | Пошаговой работы | Пошаговой работы | Ошибка или ловушка | Не формируется |

Таблица 17.1 (продолжение)

| Номер вектора | Реальный режим | Защищенный режим | Тип прерывания | Код ошибки |
|-----------------|--|---|----------------|------------------------|
| 2 | Сигнал на входе NMI | Сигнал на входе NMI | Ловушка | Не формируется |
| 3 | Контрольная точка | Контрольная точка | Ловушка | Не формируется |
| 4 | Переполнение | Переполнение | Ловушка | Не формируется |
| 5 | BIOS: печать экрана/нарушение границ массива | Нарушение границ массива | Ошибка | Не формируется |
| 6 | Недопустимая команда | Недопустимая команда | Ошибка | Не формируется |
| 7 | Обращение к отсутствующему сопроцессору | Обращение к отсутствующему сопроцессору | Ошибка | Не формируется |
| 8 | Микросхема таймера/двойная ошибка | Двойная ошибка | Авария | Формируется (всегда 0) |
| 9 | Клавиатура | Не используется | — | Формируется |
| 10 (0Ah) | Сигнал на линии IRQ_2 | Ошибкаочный TSS | Ошибка | Формируется |
| 11 (0Bh) | Сигнал на линии IRQ_3 | Отсутствие сегмента | Ошибка | Формируется |
| 12 (0Ch) | Сигнал на линии IRQ_4/выход за пределы стека | Выход за пределы стека или отсутствие стека | Ошибка | Формируется |
| 13 (0Dh) | Сигнал на линии IRQ_5/нарушение общей защиты | Нарушение общей защиты | Ошибка | Формируется |
| 14 (0Eh) | Контроллер НГМД — сигнал на линии IRQ_6 | Отсутствие страницы памяти | Ошибка | Формируется |
| 15 (0Fh) | Сигнал на линии IRQ_7 | Не используется | — | — |
| 16 (10h) | BIOS: функции видеосистемы/ошибка сопроцессора | Ошибка сопроцессора | Ошибка | Не формируется |
| 17 (11h) | Используется для прерываний BIOS и DOS | Ошибка выравнивания | Ошибка | Формируется (всегда 0) |
| 18–31 (12h–19h) | Используется для прерываний BIOS и DOS | Зарезервировано | — | — |

| Номер вектора | Реальный режим | Защищенный режим | Тип прерывания | Код ошибки |
|----------------------|--|--|----------------|----------------|
| 32–255 (20h–0ffh) | Используется для прерываний BIOS и DOS | Аппаратные прерывания; прерывания, определяемые пользователем и ОС | Ловушка | Не формируется |

Обратите внимание на то, что для некоторых прерываний в столбце для реального режима наблюдается двойственность источников прерываний (два возможных источника прерывания приведены через косую черту). Это объясняется тем, что изначально в микропроцессоре i8086/88 были жестко определены всего 5 прерываний (типа исключений, см. урок 15) с номерами 0...4. В последующих моделях микропроцессора разработчики зарезервировали номера в таблице прерываний 0...31. Но, как мы узнали на уроке 15, программы BIOS тоже настроены на обработку аппаратных прерываний с определенными номерами, которые, в свою очередь, накладываются на номера исключений из диапазона 0...31. Это может быть источником конфликтов, суть которых в том, что непонятно, от какого источника пришло прерывание. Это обстоятельство нужно учитывать программисту, который самостоятельно пишет какой-либо из обработчиков прерываний для замены системного. Системные обработчики прерываний для предотвращения таких конфликтов выполняют дополнительные действия по идентификации источника прерывания и лишь затем осуществляют непосредственно обработку прерывания.

Для того чтобы разобраться со второй причиной несовместимости механизмов обработки прерываний в реальном и защищенном режимах, рассмотрим схему обработки прерывания в защищенном режиме (рис. 17.1).

Ключевыми компонентами в этой схеме являются *дескрипторная таблица прерываний IDT* и системный регистр *idtr*. При возникновении прерывания от источника с номером *N* микропроцессор, находясь в защищенном режиме, выполняет следующие действия (см. рис. 17.1):

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

Это очень обобщенная схема. На практике все гораздо глубже, интереснее и сложнее. Начнем с того, что в защищенном режиме прерывания и исключения можно разделить на несколько групп.

- сбой;
- ловушка;
- аварийное завершение.

Это деление производится в соответствии со следующими признаками:

- какая информация сохраняется о месте возникновения прерывания или исключения?
- возможно ли возобновление прерванной программы?

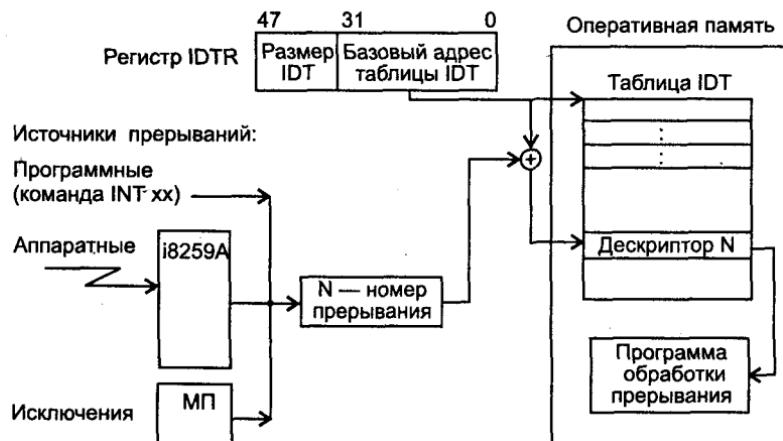


Рис. 17.1. Схема обработки прерывания в защищенном режиме

Исходя из этих признаков, можно дать следующие характеристики вышеперечисленным группам:

- Сбой (ошибка)** — прерывание или исключение, при возникновении которого в стек записываются значения регистров $cs : ip$, указывающие на команду, вызвавшую данное прерывание. Это позволяет, получив доступ к сегменту кода, исправить ошибочную команду в обработчике прерывания и, вернув управление программе, фактически осуществить ее рестарт (вспомните, что в реальном режиме при возникновении прерывания в стеке всегда запоминается адрес команды, следующей за той, которая вызвала это прерывание).
- Ловушка** — прерывание или исключение, при возникновении которого в стек записываются значения регистров $cs : ip$, указывающие на команду, следующую за командой, вызвавшей данное прерывание. Так же, как и в случае ошибки, возможен рестарт программы. Для этого необходимо лишь исправить в обработчике прерывания соответствующие код или данные, послужившие источником ошибки. После этого перед возвратом управления нужно скорректировать значение ip в стеке на длину команды, вызвавшей данное прерывание. Как видим, механизм ловушек похож на механизм прерываний в реальном режиме, хотя не во всем. Здесь есть один тонкий момент. Если прерывание типа ловушки возникло в команде передачи управления jmp , то содержимое пары $cs : ip$ в стеке будет отражать результат этого перехода, то есть соответствовать команде назначения.

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

Микропроцессор жестко определяет, какие прерывания являются ошибками, ловушками и авариями. Из приведенных выше описаний этих типов прерывания видно, что соответствующие программы-обработчики будут отличаться алгоритмами работы. По этой причине полезной является информация, приведенная в четвертом столбце табл. 17.1. В ней каждое из прерываний защищенного режима классифицировано в соответствии с приведенной выше схемой.

Но в табл. 17.1 присутствует еще один столбец. Его наличие объясняется существованием еще одной особенности: некоторые прерывания при своем возникновении дополнительно генерируют и записывают в стек так называемый *код ошибки*. Этот код может впоследствии использоваться для установления источника прерывания. Код ошибки, если он генерируется для данного прерывания (см. табл. 17.1), записывается в стек вслед за содержимым регистров *eFlags*, *cs* и *eip* (рис. 17.2).

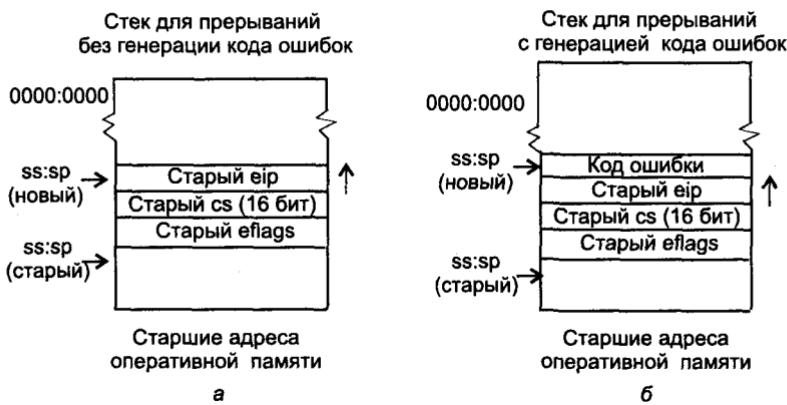


Рис. 17.2. Стек для прерываний с кодом и без кода ошибки

Какую информацию можно извлечь из кода ошибки? Код ошибки структурно представляет собой совокупность четырех полей и имеет размерность 16 бит. Если установлен режим адресации в сегменте *use32*, то при помещении в стек код ошибки расширяется нулями до 32 бит в сторону старших разрядов. Поля кода ошибки перечислены в табл. 17.2.

Теперь вам понятно, каким образом обработчик прерывания может распознать, что послужило истинной причиной прерывания, в случае, если возможна неоднозначность (см. табл. 17.1).

Что представляет собой *дескрипторная таблица прерываний IDT*?

Посмотрите еще раз на рис. 17.1. На нем показаны роль и, соответственно, функциональное назначение таблицы прерываний защищенного режима IDT. Она связывает каждый вектор прерывания с дескриптором процедуры или задачи, кото-

рая будет обрабатывать это прерывание. Формат таблицы IDT подобен формату GDT и LDT, то есть представляет собой совокупность 8-байтовых дескрипторов.

Таблица 17.2. Структура кода ошибки

| Номер бита | Обозначение | Назначение и содержимое |
|------------|-------------|--|
| 0 | EXTENT | Если EXTENT = 1, ИС вызвана аппаратным прерыванием; если EXTENT = 0, ИС вызвана последней выполнившейся командой |
| 1 | IDT | Если IDT = 1, в поле INDEX содержится индекс дескриптора в таблице IDT; если IDT = 0, в поле INDEX содержится индекс дескриптора в таблицах GDT или LDT |
| 2 | TI | Если TI = 0, значение в поле INDEX соответствует номеру дескриптора в таблице GDT; если TI = 1, значение в поле INDEX соответствует номеру дескриптора в таблице LDT |
| 3..15 | INDEX | Номер дескриптора в одной из таблиц IDT, GDT или LDT в соответствии с состояниями полей EXTENT, IDT или TI; 0 – для некоторых исключений |

Отличия таблицы IDT от указанных таблиц состоят в следующем:

- нулевой дескриптор, в отличие от таблицы GDT, используется; он описывает шлюз для программы обработки исключительной ситуации 0 (ошибка деления);
- дескрипторы в таблице IDT строго упорядочены в соответствии с номерами прерываний. В таблицах GDT и LDT, как помните, порядок описания дескрипторов роли не играет, хотя и допускается наличие некоторых соглашений по их упорядоченности;
- размерность таблицы IDT – не более 256 элементов размером по восемь байт, по числу возможных источников прерываний. В отдельных случаях есть смысл описывать все 256 дескрипторов этой таблицы, формируя для неиспользуемых номеров прерываний шлюзы-заглушки. Это позволит корректно обрабатывать все прерывания, даже если они и не планируются к использованию в данной задаче. Если этого не сделать, то при незапланированном прерывании с номером, превышающим пределы IDT для данной задачи, будет возникать исключительная ситуация общей защиты (с номером 13 (0Dh));
- при работе с прерываниями микропроцессор всегда определяет местоположение таблицы IDT по полю базового адреса в регистре idtr (см. табл. 17.2). Это он делает независимо от режима, в котором работает. В реальном режиме содержимое поля базового адреса в регистре idtr равно нулю, поэтому работа с таблицей векторов прерываний выполняется без проблем (структура ее в данном случае не имеет значения). Из вышесказанного следует, что возможно произвольное размещение в памяти этой таблицы не только в защищенном режиме, но и в реальном.

При описании этих отличий вы встретили новое понятие — *шлюз*. Так обычно называют дескрипторы в таблице прерываний IDT для того, чтобы подчеркнуть их специфику по сравнению с дескрипторами в таблицах GDT и LDT. Шлюзы предназначены для указания точки входа в программу обработки прерывания. В дескрипторной таблице прерываний IDT могут содержаться шлюзы трех типов. Их существование объясняется наличием разных по характеру источников прерывания и особенностями передачи управления в программу обработки. По формату шлюзы похожи на дескрипторы в таблицах GDT и LDT. Физически микропроцессор отличает их по значению в поле типа и содержимому остальных полей. Возможны три типа шлюзов:

- шлюз ловушки;
- шлюз прерывания;
- шлюз задачи.

Дадим им более подробную характеристику.

Шлюз ловушки

Шлюз ловушки — элемент таблицы IDT, имеющий формат, показанный на рис. 17.3.

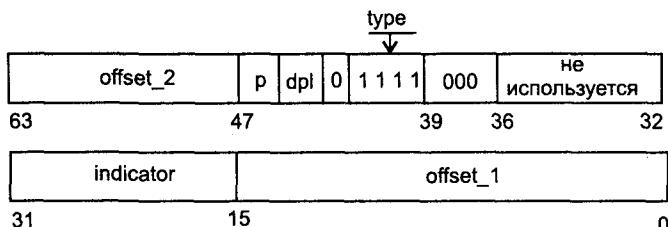


Рис. 17.3. Формат шлюза ловушки

В табл. 17.3 приведены назначения полей в формате шлюза ловушки.

Когда возникает прерывание, и его вектор выбирает в таблице IDT дескриптор шлюза с типом ловушки, микропроцессор сохраняет в стеке информацию о месте, где он прервал работу текущей программы (см. рис. 17.2). После этого он передает управление в соответствии с полями *indicator* и *offset*. Поле *indicator* представляет селектор одной из таблиц, GDT или LDT, в зависимости от состояния бита T1 в нем. Поле *offset* определяет смещение в сегменте кода. Этот сегмент кода описывается дескриптором, на который указывает селектор в поле *indicator*. После того как управление было передано обработчику прерывания, он выполняет свою работу до тех пор, пока не встретит команду *iret*. Эта команда восстанавливает из стека состояние регистров *eFlags*, *CS* и *IP* на момент возникновения прерывания, и работа приостановленной программы продолжается. При подготовке выхода из программы обработки прерывания имейте в виду, что команда *iret* ничего не знает о возможности наличия в стеке кода ошибки, поэтому для кор-

ректного возврата управления не забудьте при необходимости предварительно удалить командой `pop` код ошибки из стека.

Таблица 17.3. Поля шлюза ловушки

| Номера битов | Обозначение | Значение | Назначение |
|--------------|-------------|-----------------------------|---|
| 0...15 | offset_1 | nn...nn | Смещение в сегменте (первая половина) |
| 16...31 | indicator | mm...mm | Селектор, указывающий на дескриптор в LDT или GDT |
| 32...36 | | — | Не используется |
| 37...39 | | 000 | Постоянное значение |
| 40...43 | type | 1111 | Тип шлюза — ловушка |
| 44 | | 0 | Постоянное значение |
| 45...46 | dpl | nn (обычно dpl = 112) | Определение минимального уровня привилегий задачи, которая может передать управление обработчику прерываний через данный шлюз |
| 47 | p | 0 или 1 | Бит присутствия |
| 48...63 | offset_2 | nn...nn | Смещение в сегменте (вторая половина) |

Шлюз прерывания

Шлюз прерывания — элемент таблицы IDT, имеющий формат, показанный на рис. 17.4.

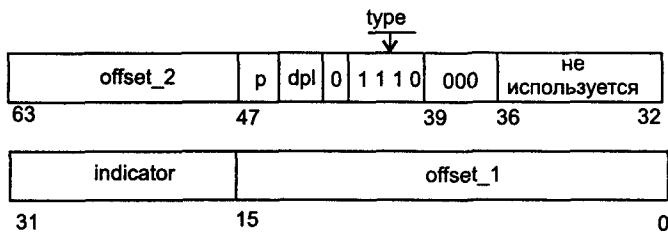


Рис. 17.4. Формат шлюза прерывания

Если внимательно посмотреть на этот формат, то можно сделать вывод, что его отличие по сравнению с форматом шлюза ловушки только в поле типа (`type`). Это имеет свою причину. При возникновении прерывания, которому соответствует шлюз прерывания, микропроцессор выполняет те же действия, что и для шлюза ловушки, но с одним важным отличием. Когда микропроцессор передает управление обработчику прерывания через шлюз прерывания, то он сбрасывает флаг прерывания в регистре `eflags` в 0, запрещая тем самым обработку аппаратных прерываний. Этот факт имеет важное значение для программирования обработчиков аппаратных и программных прерываний (см. обсуждение прерываний реального

режима в уроке 15). Если у вас есть сомнение в том, какой из двух рассмотренных типов шлюзов использовать, — применяйте шлюз прерывания.

Шлюз задачи

Шлюз задачи — элемент таблицы IDT, имеющий формат, показанный на рис. 17.5.

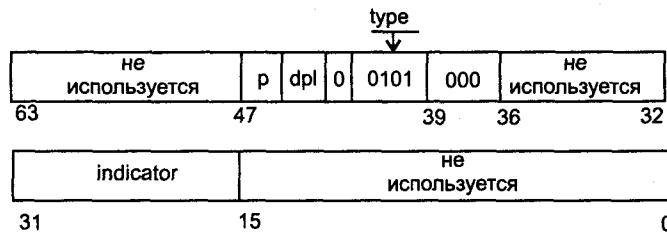


Рис. 17.5. Формат шлюза задачи

Когда микропроцессор при выполнении прерывания встречает в таблице IDT шлюз задачи, это означает, что он должен осуществить переход на особый объект. С подобными объектами, называемыми *задачами*, мы пока еще не встречались. Задача является частью механизма многозадачности. На самом деле, многозадачность не означает того, что в некоторый момент времени микропроцессор может одновременно выполнять несколько командных потоков. Микропроцессор занят в каждый конкретный момент времени выполнением команд только одного потока. Поэтому многозадачность в случае нашего микропроцессора на самом деле является псевдомногозадачностью. Это означает, что микропроцессор, выполняя команды одного потока, по истечении некоторого времени должен переключиться на выполнение команд другого потока и т. д. За счет высокого быстродействия микропроцессора создается иллюзия того, что несколько задач выполняются одновременно. Но как быть с ресурсами, которые эти потоки команд разделяют? Такими ресурсами являются, в частности, регистры. Их нужно где-то сохранять на то время, пока работает другая задача. В микропроцессорах Intel эта проблема решена следующим образом. Для каждой задачи определяется *сегмент состояния задачи* (TSS, Task Segment Status) со строго определенной структурой. В этом сегменте есть поля для сохранения всех регистров общего назначения, некоторых системных регистров и другой информации. Всю совокупность этой информации называют *контекстом задачи*. Этот сегмент описывается, подобно другим сегментам, дескриптором в таблице GDT или LDT. Если с помощью некоторого селектора обратиться к такому дескриптору, то микропроцессор осуществит переключение на соответствующую задачу. Подобные переключения могут, в частности, осуществляться операционной системой, поддерживающей многозадачность, в соответствии с некоторой дисциплиной разделения времени между задачами. Переключение задач может производиться обычными командами межсегментной передачи управления либо по возникновению прерывания при переходе к обработчику прерывания через шлюз задачи.

Таким образом, наличие дескриптора с типом шлюза задачи в таблице IDT позволяет при возникновении соответствующего прерывания переключаться на не-

которую задачу, которая будет осуществлять обработку прерывания. Поле `indicator` в шлюзе задачи содержит селектор, который определяет местонахождение дескриптора сегмента TSS — в таблице GDT или LDT (в зависимости от состояния бита TI селектора). В этой книге мы, к сожалению, не имеем возможности более подробно рассмотреть реализацию многозадачности для микропроцессоров Intel.

Итак, еще раз подчеркнем отличия перечисленных типов шлюзов. Шлюзы ловушки и прерывания с помощью полей `indicator` и `offset` определяют адрес, по которому находится точка входа в программу обработки прерывания. Шлюз задачи предназначен для реализации принципиально иного перехода к обработчику прерываний — с использованием механизма переключения задач.

После такого подробного обсуждения мы готовы к тому, чтобы рассмотреть, каким образом практически реализуется обработка прерываний в защищенном режиме. Так как при этом возникает достаточно много нюансов, то мы рассмотрим задачу, которая бы отражала суть большинства из них. В нашей программе мы будем обрабатывать одно аппаратное прерывание (от таймера), две разнотипные исключительные ситуации и обычное пользовательское прерывание. На примере этой задачи будут показаны все аспекты обработки прерываний в защищенном режиме, за исключением использования задач в качестве обработчика прерываний (см. выше обсуждение шлюзов задач).

Вначале, так же, как и для программы перехода в защищенный режим, обсудим некоторые ключевые моменты. Наша программа по-прежнему является операционной мини-системой, так как она сама обеспечивает свое функционирование на «голом» микропроцессоре. Поэтому за основу мы возьмем программу прошлого урока и дополним ее функционально, научив обрабатывать некоторые прерывания.

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

- инициализировать таблицу IDT;
- составить процедуры обработчиков прерываний;
- запретить аппаратные прерывания;
- перепрограммировать контроллер прерываний i8259A;
- загрузить регистр IDTR адресом и размером таблицы IDT;
- перейти в защищенный режим;
- разрешить обработку прерываний.

Далее микропроцессор, находясь в защищенном режиме, может производить обработку необходимых прерываний. По окончании работы для возврата в реальный режим нужно будет все «поставить на свои места», выполнив для этого последовательность следующих действий:

- запретить аппаратные прерывания;
- выполнить обратное перепрограммирование контроллера прерываний;
- перейти в реальный режим работы микропроцессора;
- разрешить обработку аппаратных прерываний.

Обсудим эти действия подробнее.

Инициализация таблицы IDT

Ранее мы уже определились с тем, что так же, как и в реальном режиме, все прерывания защищенного режима имеют свои номера от 0 до 255. Отличие их от прерываний реального режима в том, что под цели микропроцессора (исключительные ситуации, или просто исключения) отдано гораздо большее количество номеров — первые 32 вектора с номерами 0...31. Из этих 32 векторов реально используются только 0...17, остальные вектора 18...31 пока зарезервированы для будущих разработок. Для того чтобы сформировать таблицу IDT в целом, необходимо определиться с описанием отдельных ее дескрипторов — шлюзов. Можно предложить следующий, оформленный в виде структуры, шаблон для описания дескрипторов в таблице IDT:

```
descr idt struc
offs_1    dw      0          ;младшая часть адреса смещения обработчика прерывания
seldw     30          ;селектор на сегмент команд в таблице GDT
no_use    db      0          ;
type_attr db      8eh        ;по умолчанию шлюз прерывания,
                           ;для ловушки - 8fh
offs_2    dw      0          ;старшая часть адреса смещения обработчика прерывания
ends
```

Теперь можно приступить к формированию таблицы IDT. Это можно делать как в отдельном сегменте, так и в сегменте данных. Чтобы не загромождать сегмент данных, оформим таблицу IDT в виде отдельного сегмента.

```
idt_seg segment
int00h  descr_idt   <dummy_err,...,>
       rept 5
       descr_idt   <dummy,...,>
       endm
int05h  descr_idt   <int_05h,...,>
       rept 7
       descr_idt   <dummy_err,...,>
       endm
int0dh  descr_idt   <int_0dh,...,>
       rept 3
       descr_idt   <dummy,...,>
       endm
int11h  descr_idt   <dummy_err,...,>
       rept 14
       descr_idt   <dummy,...,>
       endm
int20h  descr_idt   <new_08h,...,>
int21h  descr_idt   <sirena,38,...,>
       rept 221
       descr_idt   <dummy,...,>
       endm
idt_size = $-int00h-1
idt_seg ends
```

Как видите, мы полностью описали всю таблицу прерываний. При этом мы инициализировали все дескрипторы именем процедур обработки прерываний. Основ-

ная часть дескрипторов имеет в качестве обработчиков процедуру `dummy`, то есть они будут обрабатываться одной программой. Для некоторых прерываний мы ввели уникальные имена; эти прерывания будут иметь свои процедуры обработки в нашей программе.

Обработчики прерываний

Расположение и порядок следования процедур обработки прерываний в памяти может быть произвольным. Главное, не забывать поместить их адреса в соответствующие дескрипторы.

При написании самих программ обработки прерываний мы должны помнить о том, что при возникновении некоторых прерываний в стек вслед за содержимым регистров `esp`, `cs` и `eflags` может записываться еще и *код ошибки*. Поэтому в общем случае процедура обработки прерывания должна происходить по следующей схеме действий:

1. Снять со стека и проанализировать код ошибки (если он есть).
2. Сохранить в стеке используемые в обработчике регистры микропроцессора.
3. Выполнить необходимые действия, в том числе подготовить возможный рестарт команды, вызвавшей прерывание. Подобный рестарт подразумевает возобновление выполнения прерванной программы, начиная с команды, инициировавшей процесс прерывания.
4. Восстановить сохраненные на шаге 2 регистры.
5. Выдать команду `iret`.

Прерываний, для которых микропроцессор формирует код ошибки, немного — всего 8. В программе (см. текст программы `prg17_1.asm` на диске в каталоге данного урока) приведены все возможные типы прерываний защищенного режима. Так, для исследования исключений типа ошибок взяты исключения 5 и 13 (`0Dh`), а для ловушек — 3 и обработка прерывания пользователя.

Программирование контроллера прерываний 8259A

На уроке 15 мы подробно разобрались с системой прерываний микропроцессора. Выяснили, что при загрузке BIOS производит инициализацию контроллеров прерываний (ведущего и ведомого). При этом BIOS назначает им базовые номера: ведущему — `08h`, ведомому — `70h`. Из этого следует, что если мы разрешим обработку аппаратных прерываний в защищенном режиме, то первое же прерывание от таймера заставит микропроцессор через таблицу IDT обратиться к процедуре `dummy`. Что же делать? Ничего не остается, как перепрограммировать ведущий контроллер на другое значение базового вектора. Ведомый контроллер перепрограммировать не нужно — значение его базового вектора ничем нам не мешает, так как оно находится в области, отведенной под прерывания пользователя.

Перепрограммировать контроллер прерываний мы уже умеем. Эту операцию мы проделывали на уроке 15. Теперь мы ее просто повторим, назначив, к примеру, значение `20h` базовому вектору ведущего контроллера.

Загрузка регистра IDTR

Аналогично таблице GDT, нужно сообщить микропроцессору, где находится таблица IDT. Это делается с помощью специально выделенного для этой цели регистра `idtr`. Таким образом, в защищенном режиме любая задача, имея достаточный уровень привилегий, может создать свою среду для обработки прерываний. Для загрузки регистра `idtr` можно использовать ту же структуру `point`, что и для регистра `gdtr`. Загрузка регистра `idtr` производится специальной командой `lidt`, которая имеет следующий формат:

`lidt адрес_48-битного поля` (Load IDT register) — загрузить регистр `idtr`.

Команда `lidt` загружает системный регистр `idtr` содержимым 6-байтного поля, адрес которого указан в качестве операнда.

```
point      struc
lmidw      0
adrdd      0
    ends
data       segment
point_idt point <idt_size,0>
...
data       ends
code       segment
...
    xor    eax,eax
    mov    ax,idt_seg
    shl    eax,4
    mov    dword ptr point_idt.adr,eax
    lgdt  pword point_idt
...
code       ends
```

После этого можно перейти в защищенный режим и разрешить аппаратные прерывания. Выполнив работу в защищенном режиме, мы готовимся к обратному переходу в реальный режим. Для этого, кроме восстановления вычислительной среды этого режима, необходимо восстановить и соответствующий механизм прерываний. Для повышения наглядности программы вы можете процесс перепрограммирования контроллера оформить в виде процедуры или макрокоманды.

Теперь можно собрать фрагменты в программу `prg17_1.asm`, ее текст находится на диске в каталоге данного урока.

Подведем некоторые итоги:

- Механизм прерываний защищенного режима гораздо более гибок, чем в реальном режиме. Механизмы прерываний реального и защищенного режимов несовместимы.
- Ключевыми компонентами при обработке прерываний защищенного режима являются таблица IDT и регистр `idtr`, в который загружаются адрес и размер таблицы IDT.

- Все прерывания защищенного режима по типу запоминаемого контекста делятся на три группы: ошибки, ловушки и аварии. От того, какого типа возникло прерывание, зависит, можно ли восстановить нормальное функционирование программы.
- Для восьми прерываний микропроцессор формирует код ошибки, по которому можно получить дополнительную информацию о характере и месте ошибки.
- Таблица IDT содержит три типа дескрипторов, которые называют шлюзами: шлюзы ловушек, прерываний и задач. Тип дескриптора, а значит, и действия микропроцессора, определяет значение в поле типа.
- При возникновении прерывания с некоторым номером n микропроцессор выбирает со смещением $n * 8$ от начала IDT шлюз и далее действует в зависимости от типа этого шлюза. Шлюзы ловушек и прерываний работают практически одинаково; отличие заключается только в том, что когда управление передается через шлюз прерывания, то микропроцессор сбрасывает в ноль флаг прерывания IF. Поэтому шлюзы прерываний обычно используются для обработки аппаратных и тех программных прерываний, для которых желательно запрещение прерываний от аппаратуры во время их работы. Шлюзы ловушек можно применять для обработки исключительных ситуаций, а также для перехода к обработчикам программных прерываний, для которых не требуется сброс флага IF.
- Шлюзы задач представляют собой совершенно иной способ перехода к процедуре обработки прерывания, основанный на механизме переключения задач.
- При разработке программы, переводящей микропроцессор в защищенный режим без поддержки операционной системы, требуется производить перепрограммирование ведущего контроллера прерываний. Это вызвано тем, что программы загрузки BIOS инициализируют базовый вектор ведущего контроллера прерываний значением 08h для реального режима работы микропроцессора. Значение базового вектора ведомого контроллера менять не нужно, так как его значение (70h) находится вне диапазона номеров исключений, зарезервированных для использования микропроцессором в защищенном режиме.

УРОК

18

Создание Windows-приложений на ассемблере

- Особенности разработки Windows-приложений
- Каркасное Windows-приложение на языке C/C++
- Каркасное Windows-приложение на ассемблере
- Средства TASM для разработки Windows-приложений
- Расширенное программирование на ассемблере для Win32 API
- Ресурсы Windows-приложений на языке ассемблера
- Меню в Windows-приложениях
- Перерисовка изображения
- Использование окон диалога
- Работа с графикой

В практической деятельности большинства программистов рано или поздно возникают проблемы, для решения которых необходимо знать форматы исполняемых файлов. Если ассемблер является отражением архитектуры компьютера, то формат исполняемого файла является отражением архитектуры операционной системы.

Этот и последующие уроки посвящены вопросам организации программирования на ассемблере для операционной системы Windows 95/98/NT. Формат исполняемого файла в этих системах отличается от формата файла операционной системы MS-DOS. Таким образом, мы будем иметь дело уже с несколькими форматами исполняемых файлов. Для повышения эффективности разработки программ на ассемблере необходимо понимать их различия.

Основная программно-аппаратная платформа, на которой работают операционные системы фирмы Microsoft, — компьютеры с архитектурой IBM PC. Номенклатура операционных систем, которые можно встретить сегодня на компьютерах данной архитектуры, следующая:

- MS-DOS;
- Windows 3.1;
- Windows 95/98;
- Windows NT;
- UNIX-подобные системы, в частности различные варианты Linux.

С точки зрения архитектуры, данные системы имеют достаточно сильные отличия. Формат исполняемого файла должен в той или иной мере отражать эти различия. В настоящее время существуют четыре формата исполняемых файлов для перечисленных операционных систем:

- .com (CP/M и MS-DOS);
- .exe (MS-DOS);
- .exe (Windows 3.1) или NE-формат исполняемых файлов;
- .exe (Windows 95/98/NT) или PE-формат исполняемых файлов;
- COFF- и ELF-форматы исполняемого файла UNIX.

Подробная информация об этих форматах находится в файле **format.doc** на диске в каталоге данного урока.

Программирование для операционной системы Windows всегда было занятием не из легких. На эту тему написано много книг. Сказать что-то новое, не находясь у первоисточника, здесь достаточно трудно, но и обойти эту тему в книге, которая,

как хотелось бы надеяться, претендует на звание серьезного труда, было бы не совсем правильно.

В подавляющем большинстве книг о программировании для Windows изложение, как правило, ведется на базе языка C/C++, реже — на базе Pascal. А что же ассемблер — в стороне? Конечно, нет! Мы не раз обращали ваше внимание на правильное понимание места ассемблера в архитектуре компьютера. Любая программа на языке самого высокого уровня в своем внутреннем виде представляет собой последовательность машинных кодов. А раз так, то всегда остается теоретическая возможность написать ту же программу, но уже на языке ассемблера. Непонимание или недооценка такой возможности приводит к тому, что достаточно часто приходится слышать фразу, подобную следующей: «Ах, опять этот ассемблер, но ведь это что-то несерьезное!» Также трудно согласиться с тезисом, который чаще всего следует вслед за этой фразой. Суть его сводится к утверждению того, что мощность современных компьютеров позволяет не рассматривать проблему эффективности функционирования программы в качестве первоочередной. Гораздо легче решить ее за счет увеличения объема памяти, быстродействия центрального процессора и качества компьютерной периферии. На этом уроке мы постараемся опровергнуть эти тезисы и показать, что ассемблер достаточно эффективно может быть использован для создания Windows-приложений.

Чем обосновать необходимость разработки Windows-приложений на языке ассемблера? Приведем следующие аргументы:

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

Разумеется, эти аргументы не следует воспринимать, как некоторую рекламную кампанию в поддержку языка ассемблера. Но нельзя забывать и о том, что существует бесконечное множество прикладных задач, ждущих своей очереди на компьютерную реализацию. Далеко не все из этих задач требуют применения тяжеловесных средств разработки, результатом работы которых являются столь же тяжеловесные исполняемые файлы. Многие прикладные задачи могут быть изящно выполнены на языке ассемблера, не теряя привлекательности, например, оконных приложений Windows.

Перед началом обсуждения поясним, в чем состоит разница между программированием для DOS и Windows. Это особенно полезно для читателей, которые прежде не сталкивались с программированием для Windows. Но хотелось бы сразу обратить внимание этих читателей на то, что материал данного урока не для начинающих Windows-программистов. Поэтому, если вы относите себя к этой категории, то обратитесь вначале к литературным источникам, которые специально

предназначены для начального изучения вопросов разработки программ для Windows. Это избавит вас от множества проблем, связанных с пониманием смысла тех или иных действий, необходимых для создания полноценного Windows-приложения.

Итак, операционные системы MS-DOS и Windows поддерживают две совершенно разные идеологии программирования. В чем разница? Программа DOS после своего запуска должна быть постоянно активной. Если ей что-то требуется, к примеру, получить очередную порцию данных с устройства ввода-вывода, то она сама должна выполнять соответствующие запросы к операционной системе. При этом программа DOS работает по определенному алгоритму, она всегда знает, что и когда ей следует делать. В Windows все наоборот. Программа пассивна. После запуска она ждет, когда ей уделит внимание операционная система. Операционная система делает это посылкой специально оформленных групп данных, называемых *сообщениями*. Сообщения могут быть разного типа, они функционируют в системе достаточно хаотично, и приложение не знает, какого типа сообщение придет следующим. Отсюда следует, что логика построения Windows-приложения должна обеспечивать корректную и предсказуемую работу при поступлении сообщений любого типа. В чем-то можно провести аналогию между механизмом сообщений Windows и механизмом прерываний в архитектуре IBM PC. Для обеспечения нормального функционирования своей программы программист должен уметь эффективно использовать функции интерфейса прикладного программирования (API, Application Program Interface) операционной системы.

Windows поддерживает два типа приложений:

- *оконное приложение* – строится на базе специального набора функций (API), составляющих графический интерфейс пользователя (GUI, Graphic User Interface). Оконное приложение представляет собой программу, которая весь вывод на экран производит в графическом виде. Первым результатом работы оконного приложения является отображение на экране специального объекта – окна. После того как окно отображено на экране, вся работа приложения направлена на то, чтобы поддерживать его в актуальном состоянии;
- *неоконное приложение*, также называемое *консольным*, представляет собой программу,рабатывающую в текстовом режиме. Работа консольного приложения напоминает работу программы MS-DOS. Но это лишь внешнее впечатление. Консольное приложение обеспечивается специальными функциями Windows.

Вся разница между двумя типами приложений Windows состоит в том, с каким типом информации они работают. Основной тип приложений в Windows – оконные, поэтому с них мы и начнем знакомство с процессом разработки программ для этой операционной системы. Порядок разработки консольных приложений будет приведен на уроке 20.

Любое оконное Windows-приложение имеет типовую структуру, основу которой составляет так называемое *каркасное приложение*. Это приложение содержит минимально необходимый программный код для обеспечения функционирования полноценного Windows-приложения. Не случайно во всех источниках в качестве первого Windows-приложения рекомендуется изучать и исследовать работу некоторого каркасного приложения, так как именно оно отражает основные особенности взаимодействия программы с операционной системой Windows. Более того, написав и отладив один раз каркасное приложение, вы будете использовать его в

далнейшем, как основу для написания любого другого, значительно более сложного Windows-приложения.

Изложение материала будем иллюстрировать программами на двух языках — C/C++ и ассемблере. Это значительно облегчит читателю понимание технологии написания Windows-приложений на ассемблере. Мы даже попытаемся выработать своего рода методику, с помощью которой можно будет любую понравившуюся нам программу на C/C++ переводить в функционально эквивалентную программу на ассемблере.

Перед началом изложения отметим некоторые его особенности.

- Теоретический и практический материал урока будет отражать особенности разработки программ для 32-разрядных операционных систем Windows, к которым относятся Windows 95/98 и Windows NT. Несмотря на то что архитектуры этих систем в большей или меньшей степени отличаются, их объединяет единый 32-разрядный программный интерфейс — Win32 API. Он представляет собой набор функций, к которым может обращаться приложение. Основная идея Win32 API — обеспечение переносимости программ между различными программно-аппаратными платформами.
- Несмотря на то что изложение будет вестись достаточно подробно, мы не сможем описать все детали процесса построения приложения Windows. Но в этом нет ничего страшного, так как в настоящее время доступно достаточно много источников, где это сделано с необходимой степенью детализации. Неподготовленному читателю можно посоветовать подобрать другой источник, где начальный уровень программирования для Windows изложен с соответствующей степенью детализации. При этом ему совсем не нужно влезать в дебри. Вполне достаточно достичь уровня понимания логики работы каркасного приложения Windows.
- Для изучения материала этого урока и его практического использования в дальнейшей работе мало иметь только один пакет TASM. Кроме него также необходимы пакеты инструментальных средств разработки приложений на языке C/C++, например, фирм Microsoft или Inprise. И в том, и в другом пакете имеются все необходимые средства для разработки Windows-приложений. Пакет TASM, в отличие от этих пакетов, не обладает всеми необходимыми средствами для разработки Windows-приложений. Поэтому программисту придется заимствовать их в том или ином виде в пакетах C/C++. На этом уроке мы будем использовать недостающие средства из пакета VC++ версии 4.0 фирмы Microsoft.

Каркасное Windows-приложение на C/C++

Пусть читатель не удивляется, что свое обсуждение мы начинаем с программы на языке C/C++. Это можно объяснить целями, которые перед нами стоят. Во-первых, нам необходимо понять общие принципы построения оконных приложений Windows. Во-вторых, мы разберемся с тем, какие средства ассемблера при этом используются. Приступить к достижению этих целей без предварительного обсуждения нецелесообразно. Давайте сделаем это мягко, через обсуждение минимального Windows-приложения на языке C/C++. В ходе его разработки мы введем

необходимую терминологию и сможем больше внимания уделить логике работы Windows-приложения, а не деталям его реализации. После этого мы с относительной легкостью разработаем эквивалентное приложение на ассемблере.

Приступая к разработке первого (и не только) Windows-приложения, важно понимать, что сам язык программирования мало влияет на его общую структуру. Это обстоятельство, кстати, и позволит нам чуть позже с относительной легкостью изменить инструментальное средство разработки Windows-приложений с C/C++ на ассемблер.

Минимальное приложение Windows состоит из трех частей:

- главной функции;
- цикла обработки сообщений;
- оконной функции.

Выполнение любого оконного Windows-приложения начинается с *главной функции*. Она содержит код, осуществляющий настройку (инициализацию) приложения в среде операционной системы Windows. Видимым для пользователя результатом работы главной функции является появление на экране графического объекта в виде окна. Последним действием кода главной функции является создание *цикла обработки сообщений*. После его создания приложение становится пассивным и начинает взаимодействовать с внешним миром посредством специальным образом оформленных данных — *сообщений*. Обработка поступающих приложению сообщений осуществляется специальной функцией, называемой *оконной*. Оконная функция уникальна тем, что может быть вызвана только из операционной системы, а не из приложения, которое ее содержит. Тело оконной функции имеет определенную структуру, о которой мы поговорим далее. Таким образом, Windows-приложение, как минимум, должно состоять из трех перечисленных элементов. В листинге 18.1 приведен вариант минимального приложения на языке C/C++.

Листинг 18.1. Каркасное Windows-приложение на языке C/C++

```
#include <windows.h>
LRESULT CALLBACK WindowProc(HWND , UINT , WPARAM , LPARAM);
char szClassWindow[] = "Каркасное приложение"; /*Имя класса окна*/
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpszCmdLine, int nCmdShow)
{
    HWND hWnd;
    MSG lpMsg;
    WNDCLASSEX wcl;
/* Определение класса окна */
    wcl.cbSize = sizeof(wcl); //длина структуры WNDCLASSEX
    wcl.style = CS_HREDRAW|CS_VREDRAW; //CS (Class Style) - стиль класса окна
    wcl.lpfnWndProc = WindowProc; //адрес функции окна
    wcl.cbClsExtra = 0; //для внутреннего использования Windows
    wcl.cbWndExtra = 0; //для внутреннего использования Windows
    wcl.hInstance = hInst;//дескриптор данного приложения
    wcl.hIcon = LoadIconA(NULL, IDI_APPLICATION); //стандартная иконка
    wcl.hCursor = LoadCursorA(NULL, IDC_ARROW); //стандартный курсор
    wcl.hbrBackground =(HBRUSH)GetStockObject (WHITE_BRUSH); //определить заполнение
    wcl.lpszMenuName = NULL; //окна белым цветом без меню.
```

```

wcl.lpszClassName = szClassWindow; //имя класса окна
wcl.hIconSm=NULL; //дескриптор маленькой иконки, связываемой с классом окна
//зарегистрировать класс окна
if (!RegisterClassEx (&wcl))
return 0;
//создать окно и присвоить дескриптор окна переменной hWnd
hWnd=CreateWindowEx(
    0, //расширенный стиль окна
    szClassWindow, //имя класса окна
    "Каркас программы для Win32 на C++.", //заголовок окна
    WS_OVERLAPPEDWINDOW, //стиль окна
    CW_USEDEFAULT, //X-координата верх. левого угла окна
    CW_USEDEFAULT, //Y-координата верх. левого угла окна
    CW_USEDEFAULT, //ширина окна
    CW_USEDEFAULT, //высота окна
    NULL, //дескриптор родительского окна
    NULL, //дескриптор меню окна
    hInst, //идентификатор приложения, создавшего окно
    NULL); //указатель на область данных приложения
//показать окно и перерисовать содержимое
ShowWindow (hWnd, nCmdShow);
UpdateWindow (hWnd);
/* запустить цикл обработки сообщений */
while (GetMessage(&lpMsg, NULL, 0, 0))
{
    TranslateMessage(&lpMsg); //разрешить использование клавиатуры
    DispatchMessage(&lpMsg); //вернуть управление Windows
}
return lpMsg.wParam;
} //конец WinMain
LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
//Функция WndProc вызывается операционной системой Windows 95
//и получает в качестве параметров сообщения из очереди
//сообщений данного приложения
{
    switch(message)
    {
        case WM_DESTROY: /* завершение программы */
            PostQuitMessage (0);
            break;
        default:
            //Сюда попадают все сообщения, не обрабатываемые в данной оконной функции.
            //Далее эти сообщения направляются обратно Windows на обработку по умолчанию
            return DefWindowProc (hWnd, message, wParam, lParam);
    }
    return 0;
}

```

Разберем более подробно суть действий, выполняемых каждым из трех элементов Windows-приложения. В листинге 18.1 видно, что минимальное Windows-приложение на языке C++ состоит из двух функций: главной — `WinMain` и оконной — `WindowProc`. Цель `WinMain` — сообщить системе о новом для нее приложении, его

свойствах и особенностях. Для достижения этой цели WinMain выполняет следующие действия:

- определяет и регистрирует класс окна приложения;
- создает и отображает окно приложения зарегистрированного класса;
- создает и запускает цикл обработки сообщений для приложения;
- завершает программу при получении оконной функцией соответствующего сообщения.

Оконная функция получает все сообщения, предназначенные данному окну, и обрабатывает их, возможно, вызывая для этого другие функции.

Видимая часть работы каркасного приложения заключается в создании нового окна на экране. Оно отвечает всем требованиям стандартного окна приложения Windows, то есть его можно развернуть, свернуть, изменить размер, переместить в другое место экрана и т. д. Для этого, как вы увидите, нам не придется написать ни строчки кода, а всего лишь удовлетворить определенные требования, предъявляемые к приложениям со стороны Windows.

Мы не будем больше обсуждать код листинга 18.1, так как это достаточно подробно и полно сделано в других источниках. Взамен этого мы пойдем по другому пути — заглянем за «фасад» приведенного Windows-приложения и посмотрим на работу, выполняемую компилятором по формированию соответствующего исполняемого кода. Причем сделать это целесообразно на двух этапах: в процессе формирования объектного кода и после формирования загрузочного модуля.

В процессе формирования объектного модуля компилятор преобразует исходный текст на языке C/C++ в эквивалентный текст на языке ассемблера. В контексте нашего обсуждения это достаточно ценная информация. Для того чтобы получить такой текст, необходимо в командной строке компилятора задать специальный ключ /FA (см. `makefile` в каталоге с программой `prg19_1.cpp`):

```
c1 /FA ... prg19_1.cpp
```

Полученный текст на ассемблере ценен тем, что в нем каждой строке исходного текста программы на C/C++ сопоставляется текст на ассемблере. В листинге 18.2 приведен фрагмент этого файла (`prg19_1.asm`), а полный его текст находится на диске в каталоге с исходным текстом программы `prg19_1.cpp`. Вы его можете более основательно проанализировать.

Листинг 18.2. Фрагмент ассемблерного представления исходного файла `prg19_1.cpp`

```
TITLE      prg19_1.cpp
.386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT      SEGMENT PARA USE32 PUBLIC 'CODE'
...
endif
PUBLIC ?szClassWindow@03PADA    ;szClassWindow
_DATA      SEGMENT
...
_DATA      ENDS
```

```
PUBLIC    WinMain@16
PUBLIC    ?WindowProc@@YGJPAUWND_@0I@Z ;WindowProc
EXTRN    _imp_GetMessageA@16:NEAR
```

```
TEXT     SEGMENT
; File prg19_1.cpp
```

```
_WinMain@16 PROC NEAR
```

```
; Страна 5
```

```
    push  ebp
    mov   ebp, esp
    sub   esp, 80      ;00000050H
    push  ebx
    push  esi
    push  edi
```

```
; Страна 10
```

```
; Страна 12
```

```
    mov   DWORD PTR_wcl$[ebp+8], OFFSET FLAT:?WindowProc@@YGJPAUWND_@0I@Z
:WindowProc
```

```
; Страна 13
```

```
    mov   DWORD PTR_wcl$[ebp+12], 0
```

```
; Страна 16
```

```
    push  32512 ; 00007f00H
    push  0
    call  DWORD PTR _imp_LoadIcon@8
    mov   DWORD PTR_wcl$[ebp+24], eax
```

```
; Страна 18
```

```
    push  0
    call  DWORD PTR _imp_GetStockObject@4
    mov   DWORD PTR_wcl$[ebp+32], eax
```

```
; Страна 24
```

```
    lea   eax, DWORD PTR_wcl$[ebp]
    push  eax
    call  DWORD PTR _imp_RegisterClassEx@4
    movzx eax, ax
    test  eax, eax
    jne   $L29705
```

```
; Страна 25
```

```
    xor   eax, eax
    jmp   $L29694
```

```
; Страна 27
```

```
$L29705:
```

```
; Страна 39
```

```
    push  0
    mov   eax, DWORD PTR_hInst$[ebp]
    push  eax
    push  0
    push  0
    push  -2147483648 ;80000000H
    push  -2147483648 ;80000000H
```

```

push -2147483648 ;80000000H
push -2147483648 ;80000000H
push 13565952 ;00cf0000H
push OFFSET FLAT:$SG29710
push OFFSET FLAT:?szClassWindow@3PADA ;szClassWindow
push 0
call DWORD PTR _imp_CreateWindowExA@48
mov DWORD PTR _hWnd$[ebp], eax
: Стока 41
mov eax, DWORD PTR _nCmdShow$[ebp]
push eax
mov eax, DWORD PTR _hWnd$[ebp]
push eax
call DWORD PTR _imp_ShowWindow@8
: Стока 42
mov eax, DWORD PTR _hWnd$[ebp]
push eax
call DWORD PTR _imp_UpdateWindow@4
: Стока 44
$L29712:
push 0
push 0
push 0
lea eax, DWORD PTR _lpMsg$[ebp]
push eax
call DWORD PTR _imp_GetMessageA@16
test eax, eax
je $L29713
: Стока 46
lea eax, DWORD PTR _lpMsg$[ebp]
push eax
call DWORD PTR _imp_TranslateMessage@4
: Стока 47
lea eax, DWORD PTR _lpMsg$[ebp]
push eax
call DWORD PTR _imp_DispatchMessageA@4
: Стока 48
jmp $L29712
$L29713:
: Стока 49
mov eax, DWORD PTR _lpMsg$[ebp+8]
jmp $L29694
: Стока 50
$L29694:
pop edi
pop esi
pop ebx
leave
ret 16 ;00000010H
WinMain@16 ENDP
-----?
WindowProc@@YGJPAUHWND__@IIJ@Z PROC NEAR ;WindowProc

```

```

; Страна 55
push ebp
mov ebp, esp
sub esp, 4
push ebx
push esi
push edi
; Страна 56
mov eax, DWORD PTR_message$[ebp]
mov DWORD PTR -4+[ebp], eax
jmp $L29719
; Страна 58
$L29723:
; Страна 59
push 0
call DWORD PTR _imp_PostQuitMessage@4
; Страна 60
jmp $L29720
; Страна 61
$L29724:
; Страна 64
mov eax, DWORD PTR _lParam$[ebp]
push eax
mov eax, DWORD PTR _wParam$[ebp]
push eax
mov eax, DWORD PTR_message$[ebp]
push eax
mov eax, DWORD PTR_hWnd$[ebp]
push eax
call DWORD PTR _imp_DefWindowProcA@16
jmp $L29718
; Страна 65
jmp $L29720
$L29719:
cmp DWORD PTR -4+[ebp], 2
je $L29723
jmp $L29724
$L29720:
; Страна 66
xor eax, eax
jmp $L29718
; Страна 67
$L29718:
pop edi
pop esi
pop ebx
leave
ret 16 :00000010H
?WindowProc@@YGJPAUWND__@I@Z ENDP      ;WindowProc
_TEXT ENDS
END

```

Беглый взгляд на код листинга 18.2 показывает, что он оформлен как полноценная программа на ассемблере. Код программы был генерирован компилятором

Visual C++ 4.0 фирмы Microsoft, поэтому транслятор MASM может сделать из него загрузочный модуль без дополнительного редактирования. Для этого фирма Microsoft даже подготовила специальный включаемый файл listing.inc (он находится в каталоге пакета VC++ 4.0: ..\Msdev\include, а также на диске, прилагаемой к книге).

Таким образом, имея исходный файл Windows-приложения на языке C/C++, можно получить текст на языке ассемблера. Теоретически, на его основе впоследствии можно сформировать функционально эквивалентный исполняемый модуль. Более того, если задаться целью, то не составит большого труда переделать его в вид, пригодный для обработки транслятором TASM 5.0.

Но не все так просто. Готов расстроить читателя, но мы изложили очень упрощенный подход к разработке Windows-приложения на ассемблере. Полученный код можно рассматривать лишь как схему (шаблон) такого приложения. Реально все несколько сложнее. Прежде чем мы объясним имеющие место проблемы, проведем еще один эксперимент — дизассемблируем исполняемый модуль программы `prg19_1.cpp`. Причем сделать это нужно тем дизассемблером, который «понимает» интерфейс Win32 API. В данной книге для этой цели был использован дизассемблер IDA¹. Дизассемблированный с его помощью файл можно сохранить как листинг (`.lst`) и как исходный текст ассемблера (`.asm`). Это говорит о том, что IDA также пытается по загрузочному модулю построить файл, пригодный для повторной трансляции. В начале дизассемблированного им текста даже приводятся рекомендации по использованию соответствующих опций транслятора TASM.

В контексте нашего изложения наибольший интерес представляет файл листинга (`.lst`), формируемый IDA. Анализ его содержимого позволяет заглянуть на кухню, где повара Транслятор и Редактор связей варят кушанье под названием Исполняемый Модуль. Это блюдо потребляется операционной системой и аппаратными ресурсами компьютера. Файл листинга в своей левой части содержит колонку с адресными смещениями команд. Все метки и символические имена в дизассемблированном тексте формируются с использованием этих смещений, поэтому файл удобно использовать для анализа.

Файл листинга IDA имеет большой размер, поэтому привести весь его текст в книге невозможно, да это и не нужно. Для нас интерес представляют лишь отдельные фрагменты этого файла (листинг 18.3). Полностью дизассемблированный код программы `prg19_1.cpp` находится на диске, прилагаемом к книге. Наверняка его изучение принесет вам определенную пользу.

Листинг 18.3. Фрагменты дизассемблированного кода каркасного Windows-приложения

: Этот файл сгенерирован интерактивным дизассемблером (IDA)
: Copyright (c) 1997 by DataRescue sprl, <ida@datarescue.com>

00401000 ;этот файл должен компилироваться следующей командной строкой: TASM /m1/m5
00401000 p386
00401000 model flat

¹ Информацию об этом дизассемблере вы можете получить по электронной почте: ida@geliosoft.msk.su и в Интернете: <http://azog.cs.msu.su>.

00401000_WinMain@16 proc near ;CODE XREF: start+146_p

....
00401000 push ebp
00401001 mov ebp, esp
00401003 sub esp, 50h
00401006 push ebx
00401007 push esi
00401008 push edi
....
00401037 push 0
00401039 call ds:LoadIconA
0040103F mov [ebp+var_38], eax
00401042 push 7F00h
00401047 push 0
00401049 call ds:LoadCursorA
....
00401072 lea eax, [ebp+var_50]
00401075 push eax
00401076 call ds:RegisterClassExA
0040107C movzx eax, ax
....
0040108E loc_40108E: ;CODE XREF:_WinMain@16+81_j
0040108E push 0
00401090 mov eax, [ebp+arg_0]
00401093 push eax
00401094 push 0
00401096 push 0
00401098 push 80000000h
0040109D push 80000000h
004010A2 push 80000000h
004010A7 push 80000000h
004010AC push 0CF0000h
004010B1 push offset unk_404034
004010B6 push offset unk_404020
004010BB push 0
004010BD call ds>CreateWindowExA
....
004010CE call ds>ShowWindow
004010D4 mov eax, [ebp+var_4]
004010D7 push eax
004010D8 call ds:UpdateWindow
004010DE
004010DE loc_4010DE: ; CODE XREF:_WinMain@16+10A_j
004010DE push 0
004010E0 push 0
004010E2 push 0
004010E4 lea eax, [ebp+var_20]
004010E7 push eax
004010E8 call ds:GetMessageA
004010EE test eax, eax
004010F0 jz loc_40110F
004010F6 lea eax, [ebp+var_20]
004010F9 push eax
004010FA call ds:TranslateMessage

```
00401100 lea    eax, [ebp+var_20]
00401103 push   eax
00401104 call   ds:DispatchMessageA
0040110A jmp    loc_4010DE
...
00401117 loc_401117: ;CODE XREF:_WinMain@16+89_j
00401117 pop    edi
00401118 pop    esi
00401119 pop    ebx
0040111A leave
0040111B retn  10h
0040111B_WinMain@16 endp
...
;оконная процедура
...
00401132 loc_401132: ;CODE XREF:.text:00401163_j
00401132 push   0
00401134 call   ds:PostQuitMessage
0040113A jmp    loc_40116E
...
0040114E push   eax
0040114F call   ds:DefWindowProcA
...
00401175 pop    edi
00401176 pop    esi
00401177 pop    ebx
00401178 leave
00401179 retn  10h
...
00401180
00401180 public start
00401180 start proc near
...
004011A6 call   ds:GetVersion      ;получить текущую версию Windows
004011A6 ;и информацию о текущей операционной платформе
...
00401205 call   ds:GetCommandLineA
00401208 mov    ds:dword_4050B0, eax
00401210 call   _crtGetEnvironmentStringsA
...
004012A0 call   ds:GetStartupInfoA
...
004012BF call   ds:GetModuleHandleA
004012C5 push   eax
004012C6 call   _WinMain@16
004012CB push   eax
004012CC call   _exit
...
0040130E pop    esi
0040130F pop    ebx
00401310 mov    esp, ebp
00401312 pop    ebp
00401313 retn
```

```

00401313 start endp :sp = -88h
00401313
.....
00401380_exit proc near ;CODE XREF: start+A9_p
00401380 ; start+14C_p
00401380
.....
00401389 call sub_4013C0 ;doexit
.....
00401391_exit endp
.....
004013C0 ; подпрограмма doexit
004013C0 : Атрибуты: Статическая библиотечная функция
004013C0 sub_4013C0 proc near ; CODE XREF:_exit+9_p
.....
004013D1 call ds:GetCurrentProcess
004013D7 push eax
004013D8 call ds:TerminateProcess
.....
00401458 push esi
00401459 call ds:ExitProcess
.....
00401462 retn
00401462 sub_4013C0 endp
.....
0040610C ; Импорт из KERNEL32.dll
0040610C
0040610C extrn GetModuleFileNameA:dword ; DATA XREF:_setargv+12_r
0040610C ;_NMSG_WRITE+77_r
.....
00406188 Импорт из USER32.dll
00406188 extrn PostQuitMessage:dword ;DATA XREF: .text:00401134_r
0040618C extrn DispatchMessageA:dword ;DATA XREF: _WinMain@16+104_r
.....
004061B4 end start

```

Текст листинга 18.3 дает богатую пищу для размышлений. Его обсуждение нужно начать с последней строки. По правилам ассемблера в последней директиве программы `end` должна стоять метка первой исполняемой команды. В нашем случае — это метка `start`. Если теперь посмотреть на место в тексте, куда она ссылается, то мы увидим, что в данной строке содержится директива `proc`, обозначающая начало процедуры с именем `start`. Можно сделать вывод, что исполнение программы начинается именно с этого места. Обратите внимание на то, что в теле процедуры `start` содержатся многочисленные вызовы функций.

В листинге 18.3 показаны только вызовы функций API, но если вы посмотрите на гибком диске полный вариант листинга 18.3, то увидите, что в нем содержатся вызовы множества других функций. Названия этих функций начинаются символом подчеркивания «`_`», в отличие от названий функций API, которым предшествует префикс переопределения сегмента, например: `ds:GetProcAddress`. Задержимся немного здесь и посмотрим внимательно на листинг 18.1, в частности нас интересуют функции, к которым идет обращение. Видно, что в исходном тексте Windows-приложения, написанном на C/C++, нет и намека на использование

каких-либо функций, за исключением функций API. Более того, вызов многих из функций API, которые присутствуют в дизассемблированном тексте, отсутствует в листинге 18.1. Отсюда следует естественный вывод о том, что их вставил компилятор C/C++ и, очевидно, они предназначены для инициализации среды, в которой будет функционировать программа. А так ли они необходимы? Забегая вперед, скажем, что нет. Но и отказаться от них нельзя, так как они, фактически, «навязываются» нам компилятором C/C++. В этом и состоит одна из главных причин того, что исполняемые модули на ассемблере по размеру в несколько раз меньше функционально эквивалентных модулей на языках высокого уровня. Можно сказать, что приложение, написанное на ассемблере, не содержит избыточного кода. Отметьте еще один характерный момент относительно функций API. Кодовый сегмент исполняемого модуля содержит только команды дальнего перехода к этим функциям. Сами же функции находятся в отдельном файле — dll-библиотеке. В конце листинга 18.3 содержится таблица, в которой описано местонахождение всех импортируемых функций API.

Проследим за действиями, которые производит код исполняемого модуля, формируемый компилятором Visual C++ для инициализации Windows-приложения. Для этого откройте файл `prg19_1.1st` на гибком диске с дизассемблированным текстом программы на C/C++ и переместитесь на начало процедуры `start`.

- Вызов функции API `GetVersion`¹ для определения текущей версии Windows и некоторой другой информации о системе. В настоящее время существует более информативная версия данной функции — `GetVersionEx`.
- Инициализация кучи. Используется как для динамического выделения памяти функциями языка C/C++, так и для организации ввода-вывода на низком уровне.
- Вызов функции API `GetCommandLineA` для получения указателя на командную строку, с помощью которой было вызвано данное приложение.
- Вызов функции API `GetEnvironmentStringsA` для получения указателя на блок с переменными окружения.
- Инициализация глобальных переменных периода выполнения.
- Вызов функции API `GetStartupInfoA` для заполнения структуры, поля которой определяют свойства основного окна приложения.
- Вызов функции API `GetModuleHandleA` для получения базового адреса, по которому загружен данный исполняемый модуль.
- Вызов функции `WinMain`. Как можно судить по способу вызова и внутреннему названию — `_WinMain@16` не является функцией API. Это локальная функция данного исполняемого модуля. Территориально она расположена в начале дизассемблированного текста (см. листинг 18.3). Далее поговорим о ней подробнее.

Главный вывод — текст процедуры `start` исполняемого модуля не соответствует исходному тексту программы на C/C++. В него добавлены локальные функции и функции Win32 API, которые вызываются в исполняемом модуле, например

¹ Описания всех упомянутых в книге функций и структур данных Win32 API приводятся лишь частично. Полную информацию о них вы можете получить в многочисленных источниках, в которых рассматриваются вопросы разработки Windows-приложений. Предпочтительно использовать первоисточник, находящийся в Интернете по адресу: <http://www.microsoft.com/msdn/>.

`GetVersion`, `GetCommandLineA` и т. д. Это и есть так называемый *стартовый код* программы. Если пойти взглядом далее по этому стартовому коду в файле `prg19_1.lst`, то можно увидеть вызов функции `_WinMain@16`. Найдем теперь тело этой функции в тексте. Даже беглое сравнение функции `_WinMain@16` (см. листинг 18.3) и функции `WinMain` (см. листинг 18.1) показывает, что мы нашли место, где содержится код ассемблера, функционально эквивалентный коду на C/C++. В частности, хорошо видно, что в `_WinMain@16` вызываются именно те функции (и никакие другие), что и в `WinMain` программы на C/C++.

Мы не будем сейчас обсуждать порядок и цель вызова каждой из этих функций, как это делалось для стартовой процедуры, по той причине, что следующий наш шаг — это создание программы на языке ассемблера. В процессе ее разработки мы и рассмотрим эти вопросы достаточно подробно. Сейчас нам важно сделать вывод, что функция `WinMain` не имеет прямого отношения к функциям Win32 API. Эта функция используется лишь для того, чтобы компилятор мог осуществить генерацию кода, выполняющего инициализацию приложения.

Кстати, если вы внимательно посмотрите листинг 18.3, то без труда обнаружите другие участки кода исполняемого файла, прямо соответствующие тексту исходного файла на C/C++. Например, в качестве упражнения найдите текст оконной функции.

В заключение обсуждения обратите внимание на код завершения программы:

```
004012C6      call  _WinMain@16
004012CB      push  eax
004012CC      call  _exit
```

Видно, что для завершения приложения вызывается процедура `_exit`. Код, который она содержит, является обязательным для корректного завершения любого Windows-приложения. Более подробно мы его обсудим при разработке каркасного Windows-приложения на языке ассемблера.

Каркасное Windows-приложение на ассемблере

Одним из главных критериев выбора языка разработки Windows-приложения является наличие в нем средств, способных поддержать строго определенную последовательность шагов. Язык ассемблера является универсальным языком и пригоден для реализации любых задач, поэтому можно смело предположить, что на нем можно написать также любое Windows-приложение. Материал, изложенный выше, наглядно это доказал. Более того, стали видны некоторые подробности кода, который должно содержать Windows-приложение на ассемблере. Но мало написать сам текст Windows-приложения, необходимо знать и уметь пользоваться средствами пакета транслятора, специально предназначенными для разработки таких приложений.

В листинге 18.4 приведен текст каркасного приложения на ассемблере, функционально эквивалентного Windows-приложению на C/C++ (см. листинг 18.1). Если не обращать внимания на особенности оформления кода, обусловленные требо-

ваниями синтаксиса ассемблера, то хорошо видно, что на уровне функций его структура аналогична Windows-приложению на C/C++, рассмотренному выше. Каркасное Windows-приложение на ассемблере содержит один сегмент данных .data и один сегмент кода .code. Сегмент стека в исходных текстах Windows-приложений непосредственно описывать не нужно. Windows выделяет для стека объем памяти, размер которого задан программистом в файле с расширением .def. Текст листинга 18.4 достаточно большой. Поэтому для обсуждения разобьем его комментариями на характерные фрагменты, каждый из которых затем поясним с необходимой степенью детализации.

Листинг 18.4. Каркасное Windows-приложение на ассемблере

```
<1> ;Пример каркасного приложения для Win32
<2> .386
<3> locals          ;разрешает применение локальных меток в программе
<4> .model flat, STDCALL      ;модель памяти flat
<5> ;STDCALL – передача параметров в стиле С (справа налево),
<6> ; вызываемая процедура чистит за собой стек
<7> include windowA.inc      ;включаемый файл с описаниями базовых структур
                                ;и констант Win32
<8> ;Объявление внешними используемых в данной программе функций Win32 (ASCII):
<9> extrn    GetModuleHandleA:PROC
<10> extrn   GetVersionExA:PROC
<11> extrn   GetCommandLineA:PROC
<12> extrn   GetEnvironmentStringsA:PROC
<13> extrn   GetEnvironmentStringsA:PROC
<14> extrn   GetStartupInfoA:PROC
<15> extrn   LoadIconA:PROC
<16> extrn   LoadCursorA:PROC
<17> extrn   GetStockObject:PROC
<18> extrn   RegisterClassExA:PROC
<19> extrn   CreateWindowExA:PROC
<20> extrn   ShowWindow:PROC
<21> extrn   UpdateWindow:PROC
<22> extrn   GetMessageA:PROC
<23> extrn   TranslateMessage:PROC
<24> extrn   DispatchMessageA:PROC
<25> extrn   ExitProcess:PROC
<26> extrn   PostQuitMessage:PROC
<27> extrn   DefWindowProcA:PROC
<28> extrn   PlaySoundA:PROC
<29> extrn   ReleaseDC:PROC
<30> extrn   TextOutA:PROC
<31> extrn   GetDC:PROC
<32> extrn   BeginPaint:PROC
<33> extrn   EndPaint:PROC
<34> ;объявление оконной функции объектом, видимым за пределами данного кода
<35> public WindowProc
<36> .data
<37> hwnd    dd    0
<38> hInst   dd    0
<39> hdc     dd    0
```

```

<40> ;lpVersionInformation OSVERSIONINFO      <?>
<41> wcl      WNDCLASSEX      <?>
<42> message   MSG      <?>
<43> ps       PAINTSTRUCT     <?>
<44> szClassName db      'Приложение Win32', 0
<45> szTitleName db      'Каркасное приложение Win32 на ассемблере', 0
<46> MesWindow    db      'Привет! Ну как вам процесс разработки приложения
                           на ассемблере?' 0

<47> MesWindowLen =      $-MesWindow
<48> playFileCreate      db      'create.wav', 0
<49> playFilePaint       db      'paint.wav', 0
<50> playFileDestroy     db      'destroy.wav', 0

<51> .code
<52> start      proc      near
<53> ;точка входа в программу:
<54> ;начало стартового кода
<55> ;вызовы расположенных ниже функций можно при необходимости раскомментировать,
<56> ;но они не являются обязательными в данной программе
<57> ;вызов BOOL GetVersionEx(LPOSVERSIONINFO lpVersionInformation)
<58> ;      push      offset lpVersionInformation
<59> ;      call      GetVersionExA
<60> ;далее можно вставить код для анализа информации о версии Windows (см. прил. 11)
<61> ;вызов LPTSTR GetCommandLine(VOID) – получить указатель на командную строку
<62> ;      call      GetCommandLineA      ;в регистре eax адрес
<63> ;вызов LPVOID GetEnvironmentStrings (VOID) – получить указатель
   ;на блок с переменными окружения
<64> ;      call      GetEnvironmentStringsA      ;в регистре eax адрес
<65> ;вызов VOID GetStartupInfo(LPSTARTUPINFO lpStartupInfo)      ;указатель
   ;на структуру STARTUPINFO
<66> ;      push      offset lpStartupInfo
<67> ;      call      GetStartupInfoA
<68> ;вызов HMODULE GetModuleHandleA (LPCTSTR lpModuleName)
<69> ;      push      NULL      ;0->GetModuleHandle
<70> ;      call      GetModuleHandleA      ;получить значение базового адреса,
<71> ;      mov      hInst, eax      ;по которому загружен модуль.
<72> ;далее hInst будет использоваться в качестве дескриптора данного приложения
<73> ;конец стартового кода
<74> WinMain:
<75> ;определить класс окна ATOM RegisterClassEx(CONST WNDCLASSEX *lpWndClassEx),
<76> ;      где *lpWndClassEx – адрес структуры WndClassEx
<77> ;для начала инициализируем поля структуры WndClassEx
<78> ;      mov      wcl.cbSize, type WNDCLASSEX      ;размер структуры
                           ;в wcl.cbSize
<79> ;      mov      wcl.style, CS_HREDRAW+CS_VREDRAW
<80> ;      mov      wcl.lpfnWndProc, offset WindowProc      ;адрес оконной
                           ;процедуры
<81> ;      mov      wcl.cbClsExtra, 0
<82> ;      mov      wcl.cbWndExtra, 0
<83> ;      mov      eax, hInst
<84> ;дескриптор приложения в поле hInstance структуры wcl
<85> ;      mov      wcl.hInstance, eax
<86> ;готовим вызов HICON LoadIcon (HINSTANCE hInstance, LPCTSTR lpIconName)

```

Листинг 18.4 (продолжение)

```
<87>      push    IDI_APPLICATION ;стандартный значок
<88>      push    0      ;NULL
<89>      call    LoadIconA
<90>      mov     wcl.hIcon, eax   ;дескриптор значка в поле hIcon
                                ;структуры wcl
<91> :готовим вызов HCURSOR LoadCursorA (HINSTANCE hInstance, LPCTSTR
;1pCursorName)
<92>          push    IDC_ARROW ;стандартный курсор - стрелка
<93>          push    0
<94>          call    LoadCursorA
<95>          mov     wcl.hCursor, eax   ;дескриптор курсора в поле hCursor
                                ;структуры wcl
<96> :определим цвет фона окна - белый
<97> :готовим вызов HGDIOBJ GetStockObject(int fnObject)
<98>          push    WHITE_BRUSH
<99>          call    GetStockObject
<100>         mov     wcl.hbrBackground, eax
<101>         mov     dword ptr wcl.lpszMenuName, 0 ;без главного меню
<102>         mov     dword ptr wcl.lpszClassName, offset szClassName ;имя
                                ;класса окна
<103>         mov     wcl.hIconSm, 0
<104> :регистрируем класс окна - готовим вызов RegisterClassExA (&wndclass)
<105>         push    offset wcl
<106>         call    RegisterClassExA
<107>         test   ax, ax ;проверить на успех регистрации класса окна
<108>         jz     end_cycl_msg ;неудача
<109> :создаем окно:
<110> :готовим вызов HWND CreateWindowExA(DWORD dwExStyle, LPCTSTR 1pClassName,
<111> ;           LPCTSTR 1pWindowName, DWORD dwStyle, int x, int y, int nWidth,
;int nHeight,
<112> ;           HWND hWndParent, HMENU hMenu, HANDLE hInstance, LPVOID
;1iParam)
<113>         push    0      ;1iParam
<114>         push    hInst :hInstance
<115>         push    NULL   :menu
<116>         push    NULL   :parent hwnd
<117>         push    CW_USEDEFAULT ;высота окна
<118>         push    CW_USEDEFAULT ;ширина окна
<119>         push    CW_USEDEFAULT ;координата у левого верхнего угла
                                ;окна
<120>         push    CW_USEDEFAULT ;координата x левого верхнего угла
<121>         push    WS_OVERLAPPEDWINDOW ;стиль окна
<122>         push    offset szTitleName ;строка заголовка окна
<123>         push    offset szClassName ;имя класса окна
<124>         push    NULL
<125>         call    CreateWindowExA
<126>         mov     hWnd, eax ;hwnd – дескриптор окна
<127> :показать окно:
<128> :готовим вызов BOOL ShowWindow( HWND hWnd, int nCmdShow )
<129>         push    SW_SHOWNORMAL
```

```
<130>             push    hwnd
<131>             call    ShowWindow
<132> ;перерисовываем содержимое окна
<133> ;готовим вызов BOOL UpdateWindow( HWND hWnd )
<134>             push    hwnd
<135>             call    UpdateWindow
<136> ;запускаем цикл сообщений:
<137> ;ГОТОВИМ ВЫЗОВ BOOL GetMessageA( LPMSG lpMsg, HWND hWnd )
<138> ;           UINT wMsgFilterMin, UINT wMsgFilterMax )
<139> cycl_msg:
<140>             push    0
<141>             push    0
<142>             push    NULL
<143>             push    offset message
<144>             call    GetMessageA
<145>             cmp     ax, 0
<146>             je     end_cycl_msg
<147> ;трансляция ввода с клавиатуры
<148> ;ГОТОВИМ ВЫЗОВ BOOL TranslateMessage( CONST MSG *lpMsg )
<149>             push    offset message
<150>             call    TranslateMessage
<151> ;отправим сообщение оконной процедуре
<152> ;ГОТОВИМ ВЫЗОВ LONG DispatchMessage( CONST MSG *lpmsg )
<153>             push    offset message
<154>             call    DispatchMessageA
<155>             jmp    cycl_msg
<156> end_cycl_msg:
<157>
<158> ;выход из приложения
<159> ;ГОТОВИМ ВЫЗОВ VOID ExitProcess( UINT uExitCode )
<160>             push    NULL
<161>             call    ExitProcess
<162> start      endp
<163> ;-----WindowProc-----
<164> WindowProc proc
<165> arg @hwnd:DWORD, @mes:DWORD, @wparam:DWORD, @lparam:DWORD
<166> uses ebx, edi, esi ;эти регистры обязательно должны сохраняться
<167> local   @hdc:DWORD
<168>             cmp     @@mes, WM_DESTROY
<169>             je     wmdestroy
<170>             cmp     @@mes, WM_CREATE
<171>             je     wmccreate
<172>             cmp     @@mes, WM_PAINT
<173>             je     wmpaint
<174>             jmp    default
<175> wmccreate:
<176> ;обозначим создание окна звуковым эффектом
<177> ;ГОТОВИМ ВЫЗОВ ФУНКЦИИ BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD
;fdwSound )
<178>             push    SND_SYNC+SND_FILENAME
<179>             push    NULL
<180>             push    offset playFileCreate
```

Листинг 18.4 (продолжение)

```
<181>             call    PlaySoundA
<182>             mov     eax, 0 ;возвращаемое значение - 0
<183>             jmp     exit_wndproc
<184> wmpaint:      push    SND_SYNC+SND_FILENAME
<185>             push    NULL
<186>             push    offset playFilePaint
<187>             call    PlaySoundA
<188>             ;получим контекст устройства HDC BeginPaint( HWND hwnd, LPPAINTSTRUCT
<189>             ;:lpPaint )
<190>             push    offset ps
<191>             push    @@hwnd
<192>             call    BeginPaint
<193>             mov     @@hdc, eax
<194>             ;выведем строку текста в окно BOOL TextOut( HDC hdc, int nXStart, int
<195>             ;:nYStart,
<196>             ;LPCTSTR lpString, int cbString )
<197>             push    MesWindowLen
<198>             push    offset MesWindow
<199>             push    100
<200>             push    10
<201>             push    @@hdc
<202>             call    TextOutA
<203>             ;освободить контекст BOOL EndPaint( HWND hWnd, CONST PAINTSTRUCT *lpPaint )
<204>             push    offset ps
<205>             push    @@hdc
<206>             call    EndPaint
<207>             mov     eax, 0 ;возвращаемое значение - 0
<208>             jmp     exit_wndproc
<209> wmdestroy:     push    SND_SYNC+SND_FILENAME
<210>             push    NULL
<211>             push    offset playFileDestroy
<212>             call    PlaySoundA
<213>             ;послать сообщение WM_QUIT
<214>             ;готовим вызов VOID PostQuitMessage( int nExitCode )
<215>             push    0
<216>             call    PostQuitMessage
<217>             mov     eax, 0 ;возвращаемое значение - 0
<218>             jmp     exit_wndproc
<219> default:       ;обработка по умолчанию
<220>             ;готовим вызов LRESULT DefWindowProc( HWND hWnd, UINT Msg,
<221>             ;:WPARAM wParam, LPARAM lParam )
<222>             push    @@lparam
<223>             push    @@wparam
<224>             push    @@mes
<225>             push    @@hwnd
<226>             call    DefWindowProcA
<227>             jmp     exit_wndproc
```

```
<229> :.... . . .  
<230> exit_wndproc:  
<231>         ret  
<232> WindowProc    endp  
<233> end           start
```

Строка 3. Все символические имена в программе на ассемблере по умолчанию являются глобальными. Задание директивы `locals` включает в трансляторе механизм контроля областей видимости имен и позволяет использовать в программе локальные имена. С механизмом контроля областей видимости имен вы сталкивались при программировании на языках высокого уровня. Наличие такого механизма – это своего рода шаг, сближающий ассемблер с этими языками. Символическим именам, которые вы хотите сделать локальными, должна предшествовать определенная последовательность не менее чем из двух символов. Эти символы задаются как параметры директивы `locals`. Если этого не сделать (как в нашем случае), то по умолчанию используется последовательность из двух символов – `@@`. В этом приложении мы сделали локальными символические имена, использующиеся в оконной функции `WindowProc`. Теперь, например, имя `@@hwnd` локально по отношению к оконной процедуре. Блоком, в пределах которого можно объявить локальные имена, может быть не только функция, но и участок программы между двумя метками.

Строка 4. Директива `.model` задает модель сегментации (`flat`) и стиль генерации (см. урок 14) кода при входе в процедуры программы и выходе из них (`stdcall`). Модель памяти `flat` обозначает плоскую модель памяти. В соответствии с этой моделью компилятор создает программу, которая содержит один 32-битовый сегмент для данных и кода программы. Код загрузочного модуля, генерируемый с опцией `flat`, будет работать на микропроцессорах i386 и старше. По этой причине директиве `.model` должна предшествовать одна из директив: `.386`, `.486` или `.586`. Указание этой модели памяти заставляет компоновщик создать исполняемый файл с расширением `.exe`. В программе с плоской моделью памяти используется адресация программного кода и данных типа `near`. Это же касается и атрибута расстояния в директиве `proc`, который также имеет тип `near`. Параметр `stdcall` определяет порядок передачи параметров через стек так, как это принято в языке C/C++, то есть справа налево. Этот параметр задает генерацию кода эпилога, который очищает стек по завершении работы процедуры (в стиле языка Pascal) от тех аргументов, которые были переданы в эту процедуру при вызове. Такая передача параметров очень удобна при разработке Windows-приложений: программист может помещать параметры в стек в прямом порядке и ему не нужно заботиться об очистке стека после окончания работы процедуры.

Строка 7. Директива `include` включает в программу файл `windowA.inc`. Для чего нужен этот файл? Вы наверняка имеете некоторый опыт разработки Windows-приложений на C/C++ и знаете, что функции Win32 API в качестве передаваемых им параметров используют множество констант и данных определенной структуры. В пакете компилятора C/C++ эти данные расположены в заголовочных файлах, совокупность которых можно рассматривать как дерево с корнем в виде файла `windows.h`. Для того чтобы эти описания стали доступны приложению на C/C++, в его начало включается директива `#include <windows.h>` (см. листинг 18.1). Windows-приложение на ассемблере также использует вызовы функций Win32 API, поэтому в нем должны быть выполнены аналогичные действия

по определению необходимых констант и структур. Но просто взять из пакета C/C++ и затем использовать в программе на ассемблере файл `windows.h` и связанные с ним файлы напрямую нельзя. Причина банальна: транслятор ассемблера не понимает синтаксиса C/C++. Что делать? Пакет TASM предоставляет в распоряжение программиста утилиты `h2ash.exe` и `h2ash32.exe`, которые должны помочь ему конвертировать содержимое включаемых файлов на языке C/C++ во включаемые файлы ассемблера. Однако эти утилиты весьма слабы и не стоит питать особых надежд на эффективность их работы. Поэтому проще формировать ассемблерный включаемый файл с описанием констант и структур Windows самостоятельно. Кстати, пакет TASM 5.0 предоставляет в распоряжение программиста вариант такого включаемого файла `win32.inc`. Его можно найти на диске в каталоге к данному уроку, но пользоваться им нужно с осторожностью, так как его содержимое не совсем актуально. Например, файл `win32.inc` содержит описание структуры `WNDCLASS`, но в нем отсутствует расширенный вариант этой структуры `WNDCLASSEX`, который используется в расширенном варианте функции `RegisterClassExA`. Как решить проблему актуальности описаний? Проще всего сделать это, используя включаемые файлы последней на текущий момент времени версии компилятора C/C++ (сейчас это, например, VC++ 6.0). Это один из основных источников подобной информации, актуальность которой, к тому же, гарантирована разработчиком компилятора, являющегося одновременно и создателем операционной системы Windows (речь идет, как вы догадались, о фирме Microsoft). Поэтому программирование на ассемблере для Windows предполагает, что вы хорошо умеете ориентироваться во включаемых файлах компилятора C/C++. Полностью пытаться конвертировать эти файлы не имеет смысла. Более правильно извлекать из них по мере необходимости описание нужных данных, приводить их в соответствие требованиям синтаксиса ассемблера и после этого записывать в некоторый свой файл, который будет играть роль, аналогичную `windows.h`. Для Windows-приложений данного урока вам предлагается вариант такого включаемого файла — `windowA.inc`. Его мы и включили в текст приложения директивой `include windowA.inc`. Файл `windowA.inc` содержится на диске, прилагаемой к книге. Вы можете взять его за основу для дальнейшей работы и по мере необходимости производить его наращивание.

Строки 9–33. Функции Win32 API, используемые в программе, должны быть объявлены внешними с помощью директивы `extrn`. Это необходимо сделать для того, чтобы компилятор мог генерировать правильный код, так как тела функций Win32 API содержатся в dll-библиотеках системы Windows.

Необходимо заметить, что в различных источниках встречаются разные названия, казалось бы, одной и той же функции. Необходимо правильно понимать их. Например, уже упомянутая функция `RegisterClass` может иметь названия `RegisterClassExA` или `RegisterClassExW`. На самом деле две последние функции являются более современными вариантами `RegisterClass`. Для разрешения подобных коллизий необходимо обращаться к первоисточникам — документации с описанием функций Win32 API (помните, что она также может оказаться устаревшей). Самый лучший источник такой информации — включаемые файлы компилятора C/C++. Например, описание функции `RegisterClass(ExA)` содержится в файле `winuser.h`. В листинге 18.5 приведены строки из этого файла.

Последние версии операционных систем Windows поддерживают две системы кодировки символов: однобайтную (ANSI) и двухбайтную (UNICODE).

Поддержка кодировки UNICODE была введена фирмой Microsoft для облегчения локализации программных продуктов на неанглоязычном рынке. Операционная система Windows NT использует только кодировку UNICODE. Операционная система Windows 95/98 имеет достаточно плохо скрытое наследство от MS-DOS и не поддерживает в своей внутренней работе кодировку UNICODE. Она поддерживается лишь на уровне функций Win32 API. Для обеспечения возможности работы с обеими кодировками программный интерфейс Win32 API имеет два варианта функций. Эти функции отличаются последним символом в названии. Если это A, то данная функция работает в кодировке ANSI, если W, то функция работает в кодировке UNICODE. Следующий момент, требующий пояснения, — это наличие суффикса Ex в названиях функций. Объяснение этому можно найти в файле, представленном листингом 18.5. Директива условной компиляции (строка 8) проверяет текущую версию операционной системы. Если это Windows 95/98 или NT, то можно использовать расширенные (с суффиксами Ex) варианты функций Win32 API. Они обладают дополнительными возможностями по сравнению со старыми вариантами функций Win32 API. Исчерпывающим источником по функциям Win32 API (и не только) является MSDN (Microsoft Developer Network) — информационная система поддержки разработчика по продуктам фирмы Microsoft, ее можно найти в Интернете по адресу <http://www.microsoft.com/msdn/>.

Листинг 18.5. Фрагмент файла winuser.h (VC++ 4.0)

```
<1> WINUSERAPI ATOM WINAPI RegisterClassA(CONST WNDCLASSA *lpWndClass);
<2> WINUSERAPI ATOM WINAPI RegisterClassW(CONST WNDCLASSW *lpWndClass);
<3> #ifdef UNICODE
<4> #define RegisterClass RegisterClassW
<5> #else
<6> #define RegisterClass RegisterClassA
<7> #endif // !UNICODE

<8> #if(WINVER >= 0x0400)
<9> WINUSERAPI ATOM WINAPI RegisterClassExA(CONST WNDCLASSEXA *);
<10> WINUSERAPI ATOM WINAPI RegisterClassExW(CONST WNDCLASSEXW *);
<11> #ifdef UNICODE
<12> #define RegisterClassEx RegisterClassExW
<13> #else
<14> #define RegisterClassEx RegisterClassExA
<15> #endif // !UNICODE
```

Строка 35. В соответствии с соглашениями операционной системы Windows оконная функция приложения должна быть видимой за пределами приложения, в котором она описана. Это связано с тем, что оконная функция вызывается самой операционной системой Windows при поступлении сообщений для данного приложения. По этой причине оконная функция нашего каркасного приложения объявлена общей (public).

Строки 36–50 содержат описание сегмента данных, в котором определяются переменные и экземпляры структур, используемые в каркасном приложении.

Упомянем еще об одном важном техническом моменте программирования для операционной системы Windows — о соглашениях именования различных про-

граммных объектов. Обозначениям в Windows придается большое значение. Это объясняется сложностью разрабатываемых систем, а также тем, что пользователи и разработчики должны понимать друг друга при создании как программных продуктов, так и документации к ним. Без согласованных правил по обозначению тех или иных объектов им не обойтись. Единых требований на этот счет нет. В настоящее время, де-факто, в качестве системы обозначений принята так называемая форма *венгерской записи*. Суть ее в том, что имена переменных предваряются одним или двумя символами, которые позволяют идентифицировать тип переменной. Кроме того, названия самих переменных формируются из строчных и прописных символов. Это делается для облегчения чтения названий переменных и логической увязки с теми объектами, которые эти переменные описывают. Например, что вы можете сказать о переменных с именами `param` и `cParam`? О переменной `param` практически ничего сказать нельзя, кроме того, что это какой-то параметр. Насчет переменной `cParam` можно предположить, что это параметр символьного типа, размер которого 1 байт. Правила кодирования имен функций просты, позволяют удобочитаемость текстов программ и хорошо применимы к именам функций Win32 API. Сравните стили написания имени функции `RegisterClassExA` и `registerclassexa`. В таблице 18.1 приведены некоторые соглашения по кодированию идентификаторов в венгерской форме записи. При желании вы можете дополнить их своими обозначениями. В четвертом столбце для некоторых типов данных приведены соответствующие им типы данных Win32. В пакете компилятора C/C++ (VC++) соответствие между этими типами устанавливается во включаемых файлах.

Таблица 18.1. Соглашения записи идентификаторов в венгерской записи (для Win32)

| Префикс | Тип Visual C++ | Размерность, байт | Тип в Win32 |
|-------------|--|-------------------|---------------|
| b или by | <code>unsigned char</code> — булевский тип, целое без знака | 1 | BYTE |
| f | <code>int</code> — булевский тип | 2 | BOOL |
| c | <code>char</code> — символ | 1 | BYTE |
| cb | Счетчик байт | | |
| dw | <code>unsigned long</code> — целое без знака | 4 | DWORD, UINT |
| fn | Функция | | |
| h | Дескриптор — <code>unsigned long</code> — целое без знака | 4 | HANDLE, HWND |
| hbr | Описатель кисти | 4 | |
| i или l | <code>int</code> — целое или <code>long</code> — длинное целое | 4 | LONG |
| lpfn | Длинный указатель на функцию | 4 | |
| n или w | <code>unsigned short</code> — короткое целое без знака | 2 | WORD |
| p или lp | <code>pointer</code> — длинный указатель | 4 | LPSTR, LPCSTR |
| s | Строка | | |
| sz или lpsz | ASCII-строка (строка, заканчивающаяся двоичным нулем) | | |
| x и y | координаты x и y | 4 | |

Не стоит принимать венгерскую форму записи, как некую догму. Это всего лишь одна из возможных систем обозначений. При желании вы можете ввести свою собственную систему. Для того чтобы заставить компилятор ассемблера различать строчные и прописные буквы, необходимо указать опцию `/m1` в командной строке.

Приступая к рассмотрению сегмента кода, обратите внимание на его начало (строка 51). В нем отсутствуют привычные команды загрузки сегментного регистра данных. Загрузчик Windows самостоятельно загружает сегментные регистры, при этом учитывается требуемая модель памяти (директива `.model`). Из материала первой части урока вы, видимо, поняли, что для запуска приложения под управлением Windows необходимо выполнить ряд шагов, содержание которых заключается в вызове ряда функций Win32 API. Перечислим их:

- выполнение стартового кода;
- выполнение главной функции (в C/C++ – вызов функции `WinMain`), которая выполняет следующие действия:
 - регистрирует класс окна;
 - создает окно;
 - отображает окно;
 - запускает цикл обработки сообщений;
 - завершает выполнение приложения;
- организация обработки сообщений в оконной процедуре.

А как вызывать функции Win32 API в программе на ассемблере? Вызов этих функций осуществляется аналогично вызову внешних функций (см. урок 14). Передача всех параметров осуществляется через стек. По этой причине важно знать размеры передаваемых величин. Здесь проявляется полезность венгерской системы обозначений. О размере переменной можно судить по ее названию. К тому же, в Win32 большинство переменных имеет размер двойного слова (4 байта). В соответствии с параметром `stdcall` директивы `.model` параметры в стек должны помещаться справа налево.

Продолжим рассмотрение каркасного Windows-приложения (см. листинг 18.4).

Стартовый код (строки 54–73)

Структура нашего каркасного Windows-приложения строится в соответствии с аналогичным приложением на C/C++ (см. листинг 18.1) и его дизассемблированным вариантом (см. листинг 18.3). Разрабатывая приложение на C/C++, мы «прячемся за спину» компилятора, доверяя ему часть работы. Программируя на ассемблере, мы лишаемся этой «широкой спины» и вынуждены всю работу делать сами. Что представляет собой эта работа, видно из листинга 18.3. Первый шаг Windows-приложения заключается в исполнении *стартового кода*. Стартовый код представляет собой последовательный вызов функций Win32 API. Вы можете экспериментировать с ними при разработке своих программ (например, для доступа к информации о параметрах командной строки или переменных окружения), но в общем случае использование большинства из них не является обязательным. Чтобы продемонстрировать это, в листинге 18.4 вызов некоторых функций стартового кода закомментирован. В программе листинга 18.4 из всего

стартового кода мы оставили лишь вызов функции `GetModuleHandleA` (ее вызов, кстати, тоже можно закомментировать без потери работоспособности программы). Она предназначена для идентификации исполняемого файла в адресном пространстве процесса. Здесь нужно кое-что пояснить.

В литературе по платформе Win32 часто встречаются такие понятия, как *процессы* и *потоки*. *Процесс* (приложение) представляет собой экземпляр программы, загруженной в память для выполнения. Процесс инертен, он просто владеет пространством памяти в 4 Гбайт. В этом пространстве содержатся код и данные, другие ресурсы, загружаемые в адресное пространство процесса. В качестве ресурсов могут быть, в свою очередь, исполняемые файлы или dll-библиотеки. Для того чтобы процесс исполнялся, в нем должен быть создан *поток*, который собственно и отвечает за исполнение кода, содержащегося в адресном пространстве процесса.

Зачем такие сложности? Дело в том, что Win32, по сравнению со своими предшественниками, кроме процессной многозадачности поддерживает еще и потоковую многозадачность, при которой в рамках одного процесса параллельно запускаются несколько потоков. Таким образом, в рамках одной программы параллельно выполняются несколько участков кода. В принципе, имея несколько процессоров, возможно реальное распараллеливание вычислительного процесса в рамках одного приложения.

Из-за того что в адресное пространство процесса можно загрузить несколько файлов, для работы с ними требуется некий механизм их однозначной идентификации. При загрузке исполняемого файла (или dll-модуля) в адресное пространство процесса ему присваивается уникальный номер — описатель экземпляра (`hInst` — Handle Instance). Этот номер передается операционной системой в качестве первого параметра `hInst` функции `WinMain` в программе на языке C/C++. Это значение используется впоследствии при вызове многих функций Win32 API, загружающих те или иные ресурсы для данной программы. Что касается значения, которое присваивается `hInst`, то оно равно базовому адресу в адресном пространстве процесса, по которому загружен данный файл .exe. Выяснить его значение можно с помощью функции `GetModuleHandle`. При вызове этой функции в качестве ее единственного параметра передается адрес ASCII-строки с именем исполняемого файла (dll-библиотеки), базовый адрес загрузки (значение `hInst`) которого мы хотим получить. Если вызвать функцию `GetModuleHandle`, передав ей значение `NULL`, то мы получим значение `hInst` текущей программы, что, кстати, и делает стартовый код (см. листинг 18.3) в программе листинга 18.1. Важно отметить, что эта идентификация актуальна только в рамках одного процесса.

В системе, как правило, существует по крайней мере несколько процессов, и каждый из них владеет своим четырехгигабайтным адресным пространством. И скорее всего, все они будут иметь исполняемые файлы с одинаковыми идентификаторами. Причина в том, что в Win32 адресные пространства процессов разделены, и каждый из этих процессов полагает, что он в системе один. Фрагмент кода с вызовом `GetModuleHandleA` будет выглядеть так:

```
.data
hinst dd 0
.
.
.code
push 0
```

```
call GetModuleHandleA  
mov hinst, eax
```

После этого кода вызывается главная функция приложения — `WinMain` (см. листинг 18.3).

Главная функция (строки 74–162)

Основная задача главной функции оконного приложения Windows состоит в выполнении правильной инициализации программы и корректном ее завершении. Правильная инициализация приложения предполагает выполнение ряда предопределенных шагов. Обсудим их подробнее.

Регистрация класса окна (строки 75–108)

Под *классом окна* понимается совокупность присущих ему характеристик, таких как стиль его границ, форм курсора, пиктограмм, цвета фона, наличия меню, адреса оконной процедуры, обрабатывающей сообщения этого окна. Класс окна можно впоследствии использовать для создания окон приложения функцией `CreateWindow`. Характеристики окна описываются с помощью специальной структуры `WNDCLASS` (или ее расширенного варианта `WNDCLASSEX` в Win32).

В сегменте данных в строке 32 листинга 18.4 определен экземпляр структуры `WNDCLASSEX` — `wc1`. Первое поле структуры `WNDCLASSEX` `cbSize` должно содержать длину структуры. Команда в строке 80 загружает в это поле соответствующее значение. В поле `style` можно определять стиль границ окна и его поведение при перерисовке. Значение стиля представляет собой целочисленное значение, формируемое из констант. Каждая константа означает некоторую предопределенную характеристику. Включаемый файл `winuser.h` компилятора VC++ содержит символические названия этих констант. Эти же константы, но уже в соответствии с требованиями синтаксиса ассемблера описаны в файле `windowsA.inc`. В строке 81 листинга 18.4 комбинация констант `CS_HREDRAW` и `CS_VREDRAW` определяет необходимость полной перерисовки окна при изменении его вертикального или горизонтального размера.

Строка 80. В поле `lpfnWndProc` записывается адрес оконной функции. С помощью этой функции все окна, созданные на основе этого класса, будут обрабатывать посланные им сообщения.

Поля `cbClsExtra` и `cbWndExtra` служат для указания количества байт, дополнительно резервируемых в структуре класса окна `WNDPROC` и структуре параметров окна, которая поддерживается внутри самой Windows. Обычно эти поля инициализированы нулевыми значениями.

Строка 85 формирует в поле `hInstance` дескриптор приложения, полученный ранее функцией `GetModuleHandleA`.

Строки 86–95. В поля `hIcon` и `hCursor` загружаются дескрипторы значка и курсора. После запуска приложения значок будет отображаться на панели задач Windows и в левом верхнем углу окна приложения, а курсор появится в области окна. Значки и курсоры представляют собой ресурсы и находятся в отдельных файлах. Windows предоставляет в распоряжение программиста ряд стандартных изображений курсоров и значков. В файле `winuser.h` содержатся символические имена констант, обозначающих стандартные курсоры и значки. Обратите внимание

на первый параметр функций `LoadCursorA` и `LoadIconA` — `hInst`. Это дескриптор приложения, содержащий базовый адрес ресурса значка или курсора, загруженных в процесс. Если используются их стандартные изображения, то параметр `hInst` равен `NULL`.

Строки 96–99. Команды в этих строках формируют поле `hbrBackground`, которое должно содержать значение дескриптора кисти. *Кисть* представляет собой ресурс в виде шаблона пикселов, которым закрашивается некоторый объект, в данном случае — фон окна приложения некоторого класса. Для получения такого дескриптора необходимо использовать функцию `GetStockObject`. В качестве параметра ей передается имя нужной кисти. В файле `wingdi.h` содержатся символические имена констант, определяющих стандартные кисти.

Строка 101. В поле `lpszMenuName` записывается указатель на ASCIIZ-строку с именем меню. Если меню не используется (как в нашем случае), то в поле записывается значение `NULL`.

Строка 103. Поле `hIconSm` можно рассматривать как альтернативу полю `hIcon`. В него помещается дескриптор значка, который будет связан с данным классом окна. Если поле `hIconSm` нулевое, то система будет использовать значок, определенный полем `hIcon`.

И последнее действие при описании класса окна — присвоение данному классу уникального имени. Это имя описано в виде ASCIIZ-строки в поле `szClassName` сегмента данных, и указатель на него формируется в поле `lpszClassName` (строка 102).

Строки 105–108. После инициализации структуры необходимо зарегистрировать класс окна в системе. Это действие выполняется с помощью функции `RegisterClassExA`, которой в качестве параметра передается указатель на структуру `WNDCLASSEX`.

Необходимо заметить, что после того, как класс окна зарегистрирован, структура `WNDCLASSEX` больше не нужна. У вас появляется возможность сэкономить немного памяти. Это можно сделать, используя предоставляемый ассемблером тип данных `объединение либо инициализируя поля структуры в стеке с последующим их выталкиванием. Здесь есть широкое поле для экспериментов. Дерзайте!`

Создание окна (строки 109–126)

После того как класс окна описан и зарегистрирован в системе, приложение на его основе может создать множество различных окон. Создание окна выполняется функцией Win32 API `CreateWindowEx`. Для этого ей нужно передать множество параметров. Назначение их мы рассмотрим ниже. В качестве результата функция возвращает уникальный дескриптор окна `hWnd`, который необходимо сохранить (строки 37, 125–126).

У читателя, видимо, возникает вопрос, почему для создания окна необходимы два шага: сначала определение класса окна, а лишь затем непосредственно его создание. Если не рассматривать этот вопрос в контексте концепции объектно-ориентированного программирования, то это очень удобно для практической работы. В качестве наглядного примера приведем кнопки редактора MS Word. На самом деле это маленькие окна, созданные на базе одного класса. Они используют одну оконную функцию, которая обрабатывает сообщения, посланные этим окнам. Каждой именно кнопке было послано сообщение, оконная функция определяет по

полю `hWnd` в структуре сообщения (см. ниже). Наличие одного класса для всех кнопок гарантирует их одинаковое поведение. В то же время, каждая из этих кнопок может отличаться внешним видом. Но это уже дополнительные свойства конкретного окна-кнопки, которые задаются при создании экземпляра окна параметрами функции `CreateWindowEx`.

Рассмотрим на примере Windows-приложения (см. листинг 18.4) задание значений параметров функции `CreateWindowEx` (строки 113–124).

Строка 113. Параметр `lpParam` используется при создании окна для передачи данных или указателя на них в оконную функцию. Делается это следующим образом. Все параметры, передаваемые функции `CreateWindowEx`, сохраняются в создаваемой Windows внутренней структуре `CREATESTRUCT`. Поля этой структуры идентичны параметрам функции `CreateWindowEx`. Указатель на структуру `CREATESTRUCT` передается оконной функции при обработке сообщения `WM_CREATE`. Сам указатель находится в поле `lpParam` сообщения. Значение параметра `lpParam` функции `CreateWindowEx` находится в поле `lpCreateParams` структуры `CREATESTRUCT`.

Строка 114. Параметр `hInst` – дескриптор приложения, создающего окно.

Строка 115. Параметр `hMenu` – дескриптор главного меню окна. Так как в данном варианте Windows-приложения меню у окна отсутствует, то параметр `hMenu` имеет нулевое значение.

Строка 116. Параметр `hWndParent` – дескриптор родительского окна. Между двумя окнами Windows-приложения можно устанавливать родовидовые отношения. Дочернее окно всегда должно появляться в области родительского окна. Так как у нас подобных отношений нет, то значение передаваемого параметра нулевое.

Строки 117–120. Параметры, загружаемые в стек этими строками, задают начальные координаты положения и размеры окна приложения на экране. Константа `CW_USEDEFAULT` (`80000000h`), определенная в файле `winuser.h`, позволяет «попросить» Windows использовать значения этих параметров по умолчанию.

Строка 121. Параметр `dwStyle` определяет стиль окна приложения. Значение этого параметра задается константой или комбинацией констант. Символические имена этих констант описаны во включаемом файле `winuser.h` (Visual C++ 4.0). В нашем приложении используется значение `WS_OVERLAPPEDWINDOW` (`00000000h`), которое задает стиль обычного перекрывающегося окна с рамкой, имеющего системное меню, кнопки свертывания, развертывания и закрытия окна в правом верхнем углу.

Строки 122 и 123. Параметры `szTitleName` и `szClassName` – их значения являются указателями на ASCII-строки (строки 44–45) с именем класса окна и текстом, помещаемым в заголовок окна.

Строка 124. Параметр `dwExStyle` позволяет задать дополнительные стили окна. В Win32 API существует две реализации функции создания окна: стандартная `CreateWindowA` и расширенная `CreateWindowExA`. Для их вызова используются одинаковые параметры, за исключением последнего – `dwExStyle`. При использовании функции `CreateWindowA` параметр в стек помещать не требуется, при использовании `CreateWindowExA` параметр `dwExStyle` должен быть помещен в стек последним. Дополнительные стили можно использовать вместе со стилями, задаваемыми параметром `dwStyle`.

Строка 125. Вызов функции `CreateWindowExA`. В качестве результата функция возвращает дескриптор окна `hWnd`. Он имеет уникальное значение и является одним

из важнейших описателей объектов приложения. Он передается как параметр многим функциям Win32 API и как значение полей в некоторых структурах. Одновременно в приложении может быть создано и совместно существовать несколько окон, поэтому с помощью значения дескриптора `hWnd` Windows однозначно идентифицирует то окно, для работы с которым вызывается та или иная функция Win32 API. Другой важный результат работы функции `CreateWindowExA` — посылка асинхронного сообщения `WM_CREATE` в оконную функцию приложения. В нашей программе обработка этого сообщения заключается в вызове функции `PlaySoundA`, которая воспроизводит звуковой файл. Более подробно о сообщениях и их обработке поговорим далее.

Отображение окна (строки 127–131)

В случае успешного выполнения функции `CreateWindowExA` требуемое окно будет создано, но пока это произойдет лишь внутри самой Windows, — на экране это новое окно пока еще не отобразится. Для того чтобы созданное окно появилось на экране, необходимо применить еще одну функцию Win32 API — `ShowWindowA`. В качестве параметров этой функции передается дескриптор `hWnd` окна, которое необходимо отобразить на экране, и константа, задающая начальный вид окна на экране. В зависимости от значения последнего параметра, окно отображается в стандартном виде, развернутом на весь экран или свернутом в значок. Описание символьических констант, задающих начальный вид окна на экране, содержится во включаемом файле `winuser.h`. В нашем Windows-приложении строки 127–131 задают отображение окна в стандартном виде.

Если окно создано, то в него нужно что-то выводить — текст или графику. Здесь есть несколько тонких моментов, выяснение которых позволит нам сразу стать выше на несколько ступеней в понимании принципов работы Windows. Поэтому задержимся здесь немного. Тем более что в литературе этому моменту уделяется в лучшем случае десяток строк, большая часть которых уходит на обсуждение функций и их параметров, а не на разъяснение сути происходящих при этом процессов. Ход наших рассуждений будем подкреплять практическими наблюдениями за работой приложения. Для этого можно использовать утилиты, которые позволяют нам наблюдать за сообщениями, циркулирующими в системе, и вызовами функций API. В качестве таких программных средств рекомендуем утилиты, входящие в комплект компиляторов для языка C/C++. Например, в состав Visual C++ входит утилита Spy++ (шпион), которая позволяет наблюдать за приложениями, находящимися в данный момент в системе. Неплохие результаты при исследовании приложений Windows дает использование дизассемблера W32Dasm, информация о котором находится по адресу <http://www/expage.com/page/w32dasm>, а также отладчик Turbo Debugger для Windows (TDW).

Если у вас нет ничего подобного под рукой, отчаяваться не стоит. Давайте проведем необходимые исследования, просто меняя логику работы приложения. Суть этой методики состоит в комбинировании последовательности выполнения функций Win32 API, воспроизводящих звуковые файлы и выводящих текст на экран. При этом мы будем перемещать те или иные фрагменты программы относительно друг друга. Но это еще не все. Само воспроизведение относительно длительных звуковых файлов выполняется в определенном режиме. Обратите внимание на значение параметра `fdwSound`, который мы передаем функции `PlaySoundA`. В на-

шем случае это комбинация двух констант, одна из которых (`SND_SYNC`) означает, что функция не должна возвращать управление до тех пор, пока не закончится воспроизведение звукового файла. Если комбинировать вывод текстового сообщения и воспроизведение звукового файла, то это дает некоторую задержку, в ходе которой глаз может воспринять момент действительного появления текстового сообщения на экране.

Используем изложенную методику для изучения проблемы вывода информации (не обязательно текста) на экран. В нашей программе мы экспериментируем с выводом некоторого сообщения на экран, взяв за основу строки 189–205. Выполним с ними несколько манипуляций и обсудим их результаты.

○ Вырежем строки 189–205 из оконной процедуры `WindowProc` и вставим их в промежутке между функциями `CreateWindowExA` и `ShowWindowA`. Перетранслируем заново исходный текст приложения, получим новый вариант исполняемого модуля и запустим его. Поведение нашей программы будет следующим: воспроизводится звуковой файл `create.wav` (что означает обработку асинхронного сообщения `WM_CREATE` оконной функцией), появляется пустое окно (отработала функция `ShowWindowA`), воспроизводится звуковой файл `paint.wav` (это означает, что в оконную функцию было передано сообщение `WM_PAINT`). На экране вы ничего не увидите, кроме пустого окна. Почему окно оказалось пустым, несмотря на то, что мы в него поместили текстовое сообщение функцией `TextOut?` О причинах сложившейся ситуации можно привести следующие соображения:

1. Работа функции `ShowWindowA` заключается лишь в отображении созданного функцией `CreateWindowExA` окна заданного класса и последующем заполнении фона этого окна цветом кисти, указанной в соответствующем поле структуры `WNDCLASS`.
2. Вывод в область окна становится возможным лишь после отображения окна на экране.
3. В системе Windows циркулируют два типа сообщений: синхронные и асинхронные. Их отличие — в приоритетах обработки. Функция `ShowWindowA` помещает в очередь сообщений приложения синхронное сообщение `WM_PAINT`. Все синхронные сообщения попадают в очередь сообщений приложения и обрабатываются в порядке очередности. Это и послужило причиной того, что текст, выведенный в окно функцией `TextOut`, не был отображен в окне немедленно. В отличие от предыдущих двух доводов, данный вывод сделать достаточно трудно, не имея дополнительных средств исследования, о которых мы говорили выше. Косвенно это можно выяснить, закомментировав в программе вызов функции `UpdateWindowA`, основным назначением которой является посылка асинхронного сообщения `WM_PAINT` в оконную функцию. Асинхронные сообщения не ставятся в общую очередь приложения и сразу передаются в оконную функцию. Функция `UpdateWindowA` как раз и предназначена для таких ситуаций, в которых необходимо обновить содержимое окна. К этому вопросу, а также к синхронным и асинхронным сообщениям, мы еще вернемся, поэтому примите пока все сказанное здесь на веру.

○ Разместим теперь строки 189–205 (при этом именно переместите их, а не просто скопируйте) из функции `WindowProc` сразу за вызовом функции `ShowWindowA`

(перед `UpdateWindowA`). Перетранслируйте заново исходный текст приложения, получите новый вариант исполняемого модуля и запустите его. Поведение программы несколько изменилось. Файл `create.wav` был воспроизведен, как обычно — после создания окна. Далее последовательно появились окно приложения и долгожданное сообщение в его области. И лишь после этого был воспроизведен звуковой файл `paint.wav`. Поведение приложения для этого варианта размещения фрагментов кода приводит к следующим выводам:

1. Функции Win32 API, выводящие что-либо в окно, должны размещаться после функций, реализующих действия по отображению окна.
2. Приложение должно самостоятельно следить за актуальностью содержимого своего окна.
3. Весь вывод на экран должен производиться в оконной функции.

Подтвердим правильность этих выводов следующими рассуждениями и экспериментами. Сверните и вновь разверните окно приложения. Вы увидите, что область окна вновь стала пустой, но при этом был воспроизведен звуковой файл `create.wav`. Исходя из того, что строки кода, воспроизводящие этот файл, находятся в оконной функции в том месте, где обрабатывается сообщение `WM_PAINT`, приходим к выводу — в очереди сообщений вновь оказалось это сообщение, и именно оно было обработано оконной функцией. Как сообщение `WM_PAINT` оказалось в очереди, если наше приложение на этот раз его туда не помещало? Сообщения `WM_PAINT` от `ShowWindowA` и `UpdateWindowA` к этому моменту уже обработаны, и управление передано циклу обработки сообщений. Ответ напрашивается сам собой: это делает сама Windows, именно она рассыпает в очереди сообщений всех приложений, окна которых отображены на экране, сообщение `WM_PAINT`. Но при этом Windows не берет на себя обязанность хранить содержимое этих окон. За актуальность содержимого окна должно отвечать само приложение. Сообщение `WM_PAINT` служит для приложения сигналом обновить либо заново восстановить содержимое своего окна. Следовательно, если мы хотим, чтобы результаты наших предыдущих выводов на экран были актуальны, необходимо после получения сообщения `WM_PAINT` перерисовать содержимое окна приложения. Последнее рассуждение доказывает тезис о том, что весь вывод на экран должен производиться в оконной функции как реакция на поступление сообщения `WM_PAINT`.

Последние рассуждения показали роль сообщений (и не только `WM_PAINT`), циркулирующих в системе. Теперь, понимая всю их важность, можно приступить к рассмотрению следующей важной части оконного Windows-приложения. При этом мы продолжим манипулировать фрагментами нашего программного кода (на этот раз звуковыми) для пояснения некоторых характерных моментов.

Цикл обработки сообщений

Сообщение в Win32 представляет собой объект особой структуры, формируемый Windows. Формирование и обеспечение доставки этого объекта в нужное место в системе позволяют управлять как работой самой Windows, так и работой загруженных Windows-приложений. Инициировать формирование сообщения может несколько источников: пользователь, само приложение, система Windows, другие приложения. Именно наличие механизма сообщений позволяет Windows реализовать многозадачность, которая при работе на одном процессоре является, конечно же, псевдомультизадачностью. Windows поддерживает очередь сообщений

для каждого приложения. Запуск приложения автоматически подразумевает формирование для него собственной очереди сообщений, даже если это приложение и не будет ею пользоваться. Последнее маловероятно, так как в этом случае у приложения не будет связи с внешним миром, и оно будет являться «вещью в себе».

Формат всех сообщений Windows одинаков и описывается структурой, шаблон которой содержится в файле `winuser.h`:

```
/*
 * Message structure
 */
typedef struct tagMSG {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMMSG;
```

В этом описании все типы данных вам знакомы, за исключением одного — `POINT`. Этот тип данных описан во включаемом файле `winddef.h` и представляет собой структуру вида:

```
typedef struct tagPOINT
{
    LONG x;
    LONG y;
} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPOINT;
```

На основе этого описания в файл `windowsA.h` помещено эквивалентное описание этой структуры в соответствии с синтаксисом ассемблера:

```
POINT      struc
x  ULONG  0
y  ULONG  0
    ends
MSGstruc
meshwnd  HWIND  0
mesUINT   ?
wParam    UINT   ?
lParam    UINT   ?
time      dd     0
POINT      struc {}
    ends
```

Поле `meshwnd` структуры `MSG` содержит значение дескриптора окна, которому предназначено сообщение. Это тот самый дескриптор, который возвращается функцией `CreateWindowExA` и, соответственно, однозначно идентифицирует окно в системе. Не забывайте, что приложение обычно имеет несколько окон, поэтому значение в поле `meshwnd` помогает приложению идентифицировать нужное окно.

В поле `mes` Windows помещает 32-битную константу — идентификатор сообщения, однозначно идентифицирующий тип сообщения. Для удобства все эти константы имеют символические имена, начинающиеся с `WM_` (Window Message). Посмотрите, каким образом они определены в файле `winuser.h`. Во включаемом

файле для программ на ассемблере `win32.inc` (и нашем варианте этого файла — `windowsA.h`) тоже содержатся некоторые из этих констант. Если в ходе работы вам понадобятся отсутствующие константы — возьмите их из включаемых файлов компилятора C/C++, исправив описание в соответствии с синтаксисом ассемблера. В программе на языке C/C++ эти константы используются в оконной функции оператором `switch` для принятия решения о том, какая из его ветвей будет исполняться. В оконной функции каркасного Windows-приложения на ассемблере (строки 163–232) этот оператор моделируется (строки 168–174) командами условного, безусловного переходов и `cmp`, в качестве второго операнда которой и выступает константа, обозначающая определенный тип сообщения.

Поля `lParam` и `wParam` предназначены для того, чтобы Windows могла разместить в них дополнительную информацию о сообщении, необходимую для его правильной обработки. Эти поля, например, используются при обработке сообщений о выборе пунктов меню или о нажатии клавиш клавиатуры.

В поле `time` Windows записывает информацию о времени, когда сообщение было помещено в очередь сообщений.

И наконец, поле `POINT` содержит координаты курсора мыши в момент помещения сообщения в очередь.

Итак, представим, что в системе произошло какое-то событие, например, некоторому приложению необходимо перерисовать свое окно, в результате чего Windows сформировала сообщение `WM_PAINT`. Данное сообщение попадает в очередь сообщений приложения, создавшего окно. Для того чтобы приложение могло обработать это (или любое другое) сообщение, ему необходимо сначала его обнаружить в очереди сообщений. С этой целью, после отображения окна на экране, программа «входит» в специальный цикл, называемый *циклом обработки сообщений* (строки 139–156). Выйти из этого цикла можно только по приходу сообщения `WM_QUIT`. В этом случае функция `GetMessageA` возвращает нулевое значение, и команда условного перехода в строке 146 передает управление на конец цикла обработки сообщений. В случае прихода других сообщений функция `GetMessageA` возвращает ненулевое значение, в результате чего и осуществляется вход непосредственно в тело цикла обработки сообщения. Функции `GetMessageA` передает несколько параметров (строки 140–143):

- `message` — указатель на экземпляр структуры `MSG` (строка 42). Во время работы `GetMessageA` извлекает сообщение из очереди сообщений приложения и на основе информации в нем инициализирует поля экземпляра структуры `message`. Таким образом, приложение получает полный доступ ко всей информации в сообщении, сформированном Windows;
- `hWnd` — в поле передается дескриптор окна, сообщения для которого должны будут выбираться данной функцией. Параметр позволяет создать своеобразный фильтр, заставляющий функцию `GetMessageA` выбирать из очереди и передавать в цикл обработки сообщений сообщения лишь для определенного окна данного приложения. Если `hWnd=NULL`, то `GetMessageA` будет выбирать из очереди сообщения для всех окон;
- `wMsgFilterMin` и `wMsgFilterMax` — значения данных параметров также позволяют создавать фильтр для выбираемых функцией `GetMessageA` сообщений. Они задают диапазон номеров сообщений (поле `mes` структуры `MSG`), которые будут выбираться из очереди функцией `GetMessageA` и передаваться в цикл обработки.

ки сообщений. Параметр `wMsgFilterMin` задает минимальное значение `mes`, а параметр `wMsgFilterMax`, соответственно, максимальное значение этого параметра.

Функция `GetMessageA` выполняет следующие действия:

- постоянно просматривает очередь сообщений;
- выбирает сообщения, удовлетворяющие заданным в функции параметрам;
- заносит информацию о сообщении в экземпляр структуры `MSG` (строка 42);
- передает управление в цикл обработки сообщений.

Цикл обработки сообщений (строки 147–155) состоит всего из двух функций: `TranslateMessage` и `DispatchMessageA`. Эти функции имеют единственный параметр — указатель на экземпляр структуры `MSG`, предварительно заполненный информацией о сообщении функцией `GetMessageA`.

Функция `TranslateMessage` предназначена для обнаружения сообщений от клавиатуры для данного приложения. Если приложение не будет самостоятельно обрабатывать ввод с клавиатуры, то эти сообщения передаются для обработки обратно Windows.

Функция `DispatchMessageA` предназначена для передачи сообщения оконной функции. Такая передача производится не напрямую, так как сама `DispatchMessageA` ничего не знает о месторасположении оконной функции, а косвенно — посредством системы Windows. При этом:

- функция `DispatchMessageA` возвращает сообщение операционной системе;
- Windows, используя описание класса окна, передает сообщение нужной оконной функции приложения;
- после обработки сообщения оконной функцией управление возвращается операционной системе;
- Windows передает управление функции `DispatchMessageA`;
- `DispatchMessageA` завершает свое выполнение.

Так как вызов функции `DispatchMessageA` является последним в цикле, то управление опять передается функции `GetMessageA`. `GetMessageA` выбирает очередное сообщение из очереди сообщений и, если оно удовлетворяет параметрам, заданным при вызове этой функции, то выполняется тело цикла. Цикл обработки сообщений выполняется до тех пор, пока не приходит сообщение `WM_QUIT`. Получение этого сообщения — единственное условие, при котором программа может выйти из цикла обработки сообщений.

Завершение выполнения приложения

Выход из цикла обработки сообщений означает одно — необходимо завершить программу. В программе на C/C++ это делается так — непосредственно за циклом обработки сообщений помещается оператор `return` (см. листинг 18.1):

```
return 1pMsg.wParam;
```

В качестве операнда в операторе `return` используется значение поля `wParam` экземпляра структуры `MSG` — `1pMsg`. Значение этого поля формируется значением соответствующего поля последнего сообщения, выбранного функцией `GetMessageA` из очереди. Нетрудно догадаться, что этим сообщением было `WM_QUIT`.

Листинг 18.3 позволяет посмотреть, каким образом процесс завершения Windows-приложения реализован компилятором языка C/C++.

```
004012C6 call _WinMain@16
004012CB push eax
004012CC call _exit
...
00401380_exit proc near ; CODE XREF: start+A9_p
...
00401389 call sub_4013C0 ; _doexit
...
00401391_exit endp
...
004013C0 ; _doexit
...
004013D1 call ds:GetCurrentProcess
004013D7 push eax
004013D8 call ds:TerminateProcess
...
00401458 push esi
00401459 call ds:ExitProcess
...
00401462 retn
```

Из полного варианта листинга 18.3 видно, что процедура `WinMain` возвращает в регистре `eax` значение `wParam` сообщения `WM_QUIT`. Затем вызывается локальная процедура `_exit`, предназначенная для выполнения определенных действий по завершению приложения. Процедура `_exit`, в свою очередь, вызывает другую локальную процедуру `_doexit`. Ее текст представляет наибольший интерес для нас, так как в нем мы видим те функции Win32 API, которые непосредственно выполняют работу по удалению приложения из системы Windows. Это три функции: `GetCurrentProcess`, `TerminateProcess` и `ExitProcess`. Для завершения работы приложения достаточно использовать только функцию `ExitProcess`, что и сделано в разработанной нами программе (строки 158–161 листинга 18.4).

На этом рассмотрение работы главной функции стандартного Windows-приложения можно считать оконченным. Видимо, вы обратили внимание, что до сих пор вся работа шла в интересах Windows: инициализировались определенные структуры данных, вызывались строго определенные функции и т. д. А где же полезная работа приложения? Выполнением этой работы занимается оконная функция. Если быть более точным, то она выступает координатором этой работы. Далее разберемся с тем, как реализовать оконную функцию при разработке Windows-приложения на языке ассемблера.

Обработка сообщений в оконной функции

Оконная функция предназначена для организации адекватной реакции со стороны Windows-приложения на действия пользователя и поддержания в актуальном состоянии того окна приложения, сообщения которого она обрабатывает. Приложение может иметь несколько оконных функций. Их количество определяется количеством классов окон, зарегистрированных в системе функцией `RegisterClass(Ex)`. Если вы помните, функции `RegisterClass(Ex)` посредством экземпляра

структуре `WNDCLASS` передается указатель на определенную оконную функцию Windows-приложения. Данная функция до конца работы приложения связана с экземплярами окон, которые, в свою очередь, создаются другой функцией API — `CreateWindowEx`.

Когда для окна Windows-приложения появляется сообщение, операционная система Windows производит вызов соответствующей оконной функции. Ранее мы для упрощения говорили, что единственный источник появления сообщений — очередь сообщений приложения, но это не совсем так. Дело в том, что сообщения, в зависимости от источника их появления в оконной функции, могут быть двух типов: синхронные и асинхронные. К *синхронным* сообщениям относятся те сообщения, которые помещаются в очередь сообщений приложения и терпеливо ждут момента, когда они будут выбраны функцией `GetMessage`. После этого поступившие сообщения попадают в оконную функцию, где и производится их обработка. *Асинхронные* сообщения, подобно пожарной машине, попадают в оконную функцию в экстренном порядке, минуя при этом все очереди. Асинхронные сообщения, в частности, инициируются некоторыми функциями Win32 API, такими как `CreateWindow(Ex)` или `UpdateWindow`. Координацию синхронных и асинхронных сообщений осуществляет Windows. Если рассматривать синхронное сообщение, то его извлечение производится функцией `GetMessage` с последующей передачей его обратно в Windows функцией `DispatchMessage`. Асинхронное сообщение, независимо от источника, который инициирует его появление, сначала попадает в Windows и затем — в нужную оконную функцию.

Для чего реализуется именно такая схема, неужели функции `DispatchMessage` нельзя сразу передать сообщение в оконную функцию? Если бы это было так, то в системе получилось бы одно приложение-монополист, которое захватило бы все процессорное время своим циклом обработки сообщений. В схеме, реализованной в Windows, обработка сообщений приложением проводится в два этапа: на первом этапе приложение выбирает сообщение из очереди и отправляет его обратно во внутренние структуры Windows; на втором этапе Windows вызывает нужную оконную функцию приложения, передавая ей параметры сообщения. Преимущество этой схемы в том, что Windows самостоятельно решает все вопросы организации эффективной работы приложений.

Таким образом, при поступлении сообщения Windows вызывает оконную функцию и передает ей ряд параметров. Все они берутся из соответствующих полей сообщения. В нотации языка C/C++ заголовок оконной функции описан следующим образом (см. листинг 18.1):

```
LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam),
```

где `hWnd` — дескриптор окна, которому предназначено сообщение; `message` — идентификатор сообщения, характеризующий тип сообщения; `wParam` и `lParam` — дополнительные параметры, являющиеся копиями соответствующих полей структуры поступившего сообщения.

Оставшиеся два поля структуры `MSG`: `time` и `POINT` используются достаточно редко, и при необходимости их значения можно извлечь непосредственно из экземпляра структуры сообщения.

В нашем примере Windows-приложения строки 164–167 обозначают начало оконной функции. Параметры этой функции Windows помещает в стек. Чтобы можно

было работать с ними, используя символические имена, после заголовка функции поместим директиву `arg` (см. урок 14).

Windows требует, чтобы оконная функция сохраняла значения регистров `esi`, `edi` и `esi`. Причина — оконная функция должна быть рекурсивной. Например, возможна ситуация, когда несколько классов окон используют одну и ту же оконную функцию для обработки сообщений, поступающих в созданные на базе этих классов окна. Имеет смысл сохранять не только вышеизложенные регистры, используемые самой Windows, но и другие регистры (за исключением `eax`), если они задействованы в оконной функции. Исходя из требования рекурсивности, все переменные, используемые в оконной функции, должны быть локальными. Чтобы удовлетворить требованию сохранения регистров, лучше использовать соответствующее средство транслятора ассемблера — директиву `uses` (строка 166 листинга 18.4, см. также урок 10). При ее использовании транслятор вставляет в начало и конец оконной функции соответствующую последовательность команд ассемблера `push`. Вы можете убедиться в этом, посмотрев листинг (файл `.lst`) приложения 18.4 после его трансляции.

Центральным местом оконной функции является синтаксическая конструкция, в задачу которой входит распознавание поступившего сообщения по его типу (параметр `message`) и передача управления на ту ветвь кода оконной функции, которая продолжает далее работу с параметрами сообщения. В языке C/C++ (см. листинг 18.1) для этого используется оператор `switch` (переключатель). В языке ассемблера такого средства нет, поэтому его приходится моделировать. В листинге 18.4 строки 168–174 показывают, как это можно сделать с использованием команды `str`, команд условного `je` и безусловного `jmp` переходов.

Соответствие символьических имен константам, обозначающим тип сообщений Windows, приведено во включенном файле `windowsA.h`. Совершенно необязательно анализировать типы всех возможных сообщений, параметры которых передаются в оконную функцию. Структуру подобную строкам 168–174, и вообще структуру всей оконной функции можно сравнить с ситом, а сообщения — с зернами разной величины. Отверстия в сите непростые — каждое из них пропускает зерно своего размера. Зерна (сообщения) в сите (оконной функции) не задерживаются, попав в него, они через одно из отверстий обязательно уходят. Уходят куда? Это опять-таки определяется особой структурой нашего сита. Представьте, что к каждому отверстию припаяна трубка, которая определяет, куда данное зернышко попадет для дальнейшей обработки. Таким образом, каждому сообщению, попадающему в оконную функцию, соответствует ветвь в коде процедуры, которая начинает обработку этого сообщения. В ходе этой обработки, возможно, понадобится вызов других функций или применение синтаксических конструкций, подобных описанному ситу, для дальнейшей, более тонкой селекции сообщений, но уже одного типа (по полям `lParam` или `wParam`). Для наглядности аналогию структуры оконной функции с ситом можно упрощенно представить в виде рис. 18.1.

Рис. 18.1 показывает, что оконная функция, обрабатывающая сообщения, имеет одну точку входа и множество точек выхода. Выход из оконной функции осуществляется из той ее ветви, где обрабатывалось сообщение. Сообщения, для которых не предусмотрена отдельная обработка, должны обрабатываться функцией `DefWindowProc` (строки 219–227). Эта процедура по отношению к переданным ей сообщениям предпринимает действия по умолчанию. В принципе, ей можно пе-

редавать на обработку все сообщения, что в контексте рассмотренной выше абстракции означает сито, содержащее одно отверстие, в которое проваливаются все поступающие сообщения.

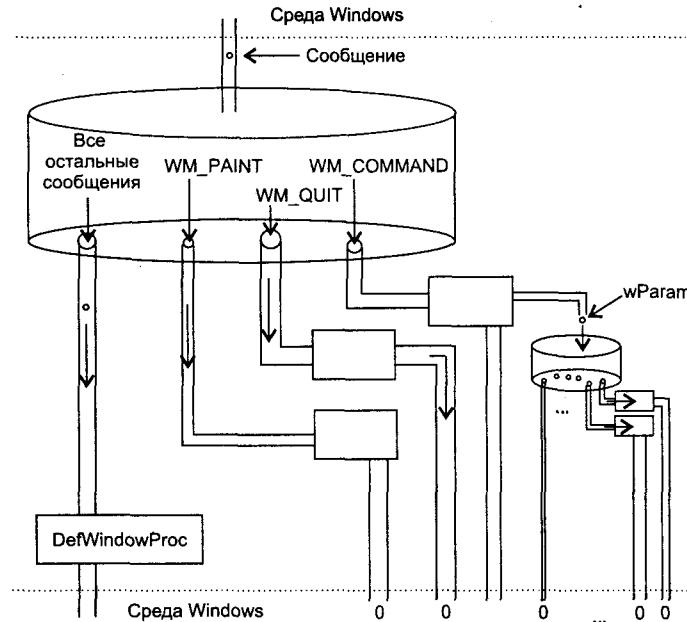


Рис. 18.1. Абстрактное представление структуры оконной функции Windows

По завершении работы оконная функция формирует значение в регистре `eax`. Если сообщение обрабатывалось в оконной функции, то в `eax` необходимо поместить нулевое значение. Если обработка осуществлялась по умолчанию, то есть функцией `DefWindowProc`, то в `eax` уже сформировано возвращаемое значение, и именно его нужно возвратить в качестве результата работы оконной функции.

В нашем примере оконная функция обрабатывает три сообщения: создание окна WM_CREATE, перерисовку области окна WM_PAINT, закрытие окна WM_DESTROY.

Более подробно с условиями возникновения и обработкой этих и других сообщений можно познакомиться в литературе по Windows, где эти вопросы освещены более полно.

Средства TASM для разработки Windows-приложений

Ранее мы подробно разобрались с тем, что собой представляет простое Windows-приложение, написанное на языке ассемблера. Излагая материал, мы упоминали имена файлов, которые нужны для получения работоспособного исполняемого модуля программы. Для устранения возможных неясностей соберем эту информацию в одном месте и систематизируем ее.

При разработке Windows-приложений на языке ассемблера с использованием Win32 API вам не обойтись без пакета TASM версии 5.0. Современные 32-разрядные операционные системы (Windows 95/98 и Windows NT) используют PE-формат исполняемого файла. В состав пакета TASM 5.0 входят два компилятора ассемблера — 16- и 32-разрядный. Они имеют имена исполняемых файлов, соответственно, `tasm.exe` и `tasm32.exe`. То же касается и редакторов связей — `tlink.exe` и `tlink32.exe`. Получить файл формата PE можно только с использованием `tasm32.exe` совместно с `tlink32.exe`. Для удобства работы скопируйте все файлы 32-разрядной версии в отдельный каталог. Сюда же необходимо поместить файлы `windowsA.inc` и `import32.lib` с дискеты, прилагаемой к книге.

Для создания программы нужны еще два файла: файл определений компоновщика и файл описания ресурсов. Файл описания ресурсов будет рассмотрен в следующем разделе. Что касается *файла определений компоновщика*, то его содержимое достаточно подробно описывается в различных источниках, и его роль ничем не отличается от роли аналогичного файла при разработке Windows-приложений на других языках. Назначение файла определений компоновщика состоит в том, чтобы предоставить редактору связей информацию о способе загрузки программы. Несмотря на то что в архитектуре Win32 нет особого смысла использовать данный файл, тем не менее, `tlink32.exe` требует указания этого файла среди файлов, подаваемых ему в качестве входных. Вы можете без проблем использовать готовый вариант этого файла на дискете.

Перечислим необходимые для разработки Windows-приложения файлы:

- файл с исходным текстом программы (`.asm`). Формируется программистом;
- включаемый файл с описаниями структур данных и констант Win32 (`.inc` или `.ash`). Файл формируется программистом по мере расширения используемых им средств Win32. Источником информации для этого файла служат включаемые файлы (`.h`) из пакета компилятора C/C++, например, VC++ версии 4.0 и выше;
- файл с библиотекой импорта `import32.lib`. Этот файл требуется компоновщику для разрешения внешних ссылок на функции Win32 API. Вы можете создать этот файл сами. Такая необходимость может возникнуть, если вам понадобится использовать функции из dll-библиотек, информация о которых отсутствует в существующем варианте файла `import32.lib`. Для этого существует специальная утилита `implib.exe`, поставляемая в пакете TASM 5.0. Командная строка для ее запуска имеет вид:

```
implib имя_файла_lib список_dll_библиотек
```

Получить информацию о местонахождении конкретной функции Win32 API достаточно просто. Многие справочные руководства по Windows при описании конкретной функции приводят и информацию о dll-библиотеке, где эта функция содержится;

- файл с описанием ресурсов, используемых в приложении;
- другие файлы. Например, в рассмотренной нами программе каркасного приложения (см. листинг 18.4) используются звуковые файлы (`.wav`);
- файлы `tasm32`, `tlink32`, `exe` и, возможно, некоторые другие вспомогательные файлы из пакета TASM 5.0. Следите внимательно за сообщениями. В том случае, если какого-либо файла будет недоставать, его нужно просто найти в катало-

ге \bin пакета TASM 5.0 и скопировать в свой рабочий каталог. Непосредственно в каталоге \bin работать не рекомендуется, иначе он моментально превратится у вас в слабоструктурируемое нагромождение файлов;

- компилятор ресурсов brc32.exe или brcc32.exe. Компиляторы взяты из пакета C/C++ фирмы Inprise. Но если вы работаете с пакетом VC++, то вам может понадобиться компилятор ресурсов, входящий в этот пакет. Он называется rc.exe;
- файл makefile и утилита make.exe. Эти файлы призваны облегчить процесс сборки приложения в единый исполняемый модуль.

Приведенный список файлов, необходимых для сборки Windows-приложения, может повергнуть в ужас читателя. До этого урока процесс получения исполняемого файла был простым и вполне управляемым из командной строки, без использования, например, make-файлов. Более сложные приложения требуют учета взаимосвязей между несколькими файлами. Данные файлы, в свою очередь, создаются или обрабатываются разными программными средствами, которые иногда требуют задания особенностей работы многочисленными опциями. Запоминать их и постоянно вводить вручную тяжело, и такая работа вряд ли может быть признана эффективной. Для облегчения процесса получения исполняемого файла используйте возможности, предоставляемые make-файлами.

Использование make-файлов для программиста несет существенное облегчение в его работе. Тщательно разработав один раз make-файл для создания исполняемого файла своего приложения, вы впоследствии избавите себя от рутинной работы по формированию необходимых для этого командных строк. Второй положительный эффект от использования make-файлов — упрощается работа автора по описанию процесса получения исполняемого модуля. Кстати, в роли автора можете оказаться и вы, когда разработаете свою программу и вместо длинного описания процесса ее сборки предоставите пользователям make-файл, дополнив его необходимыми комментариями. Каким образом написать make-файл, изложено на уроке 4. Make-файл для сборки приложения (см. листинг 18.4) представлен в листинге 18.6.

Листинг 18.6. Пример make-файла для создания приложения prg19_1.exe

```
<1> NAME = prg19_1
<2> OBJS = $(NAME).obj
<3> DEF = $(NAME).def
<4> TASMDEBUG=/zi
<5> LINKDEBUG=/v
<6> IMPORT=import32
<7> $(NAME).EXE: $(OBJS) $(DEF)
<8> tlink32 /Tpe /aa /c $(LINKDEBUG) $(OBJS), $(NAME), , $(IMPORT), $(DEF)
<9> .asm.obj:
<10> tasm32 $(TASMDEBUG) /m1 $&.asm, . . .
```

Поясним наиболее значимые элементы приведенного файла. Мы уже упоминали, что трансляция исходного файла производится программой tasm32.exe. При вызове ей передается ряд опций (строки 1–6). Для формирования отладочной информации вы можете указывать значение макрооператора TASMDEBUG=/zi (строка 4) и значение макрооператора LINKDEBUG=/v (строка 5). Кроме этих необязатель-

ных опций, в строке 10 присутствует обязательная опция /m1, которая требует, чтобы транслятор ассемблера различал строчные и прописные буквы при написании идентификаторов программы. Как вы смогли убедиться, это очень удобно при написании программ для Windows. Стока 8 содержит вызов компоновщика tlink32.exe. Описание других опций для этой программы приведены в уроке 4, также их можно найти в Справочнике.

При запуске make.exe на выполнение не забывайте о нехитром, но полезном приеме с перенаправлением вывода экранных сообщений в файл с помощью символа «>».

Углубленное программирование на ассемблере для Win32

Реализация описанного выше процесса разработки простого Windows-приложения на языке ассемблера отняла у нас много сил и времени. Но хотелось бы надеяться, что это была не пустая трата драгоценных для любого программиста жизненных сил. Цель, которая преследуется изложением всего вышеописанного материала именно в таком виде, — показать читателю, что разработка Windows-приложения на языке ассемблера — не такое уж нереальное дело. Напротив, у него даже есть свои достоинства. Нужно отметить, что объем учебного материала, необходимого для описания процесса разработки каркасного приложения для Windows, не зависит от языка, на котором предполагается вести программирование, так как основное внимание уделяется не столько средствам языка, сколько удовлетворению требований к приложению со стороны Windows. Каркасное приложение является простейшей программой для Windows, которая в лучшем случае выводит строку текста в окно приложения. С точки зрения сложности, не имеет смысла даже проводить ее сравнение с аналогичной программой для MS-DOS, выводящей строку на экран. Логику работы и способы реализации такой программы для MS-DOS можно в худшем случае объяснить минут за десять. Для объяснения логики работы каркасного Windows-приложения неподготовленному слушателю вам придется прочитать целую лекцию, может быть, и не одну. И это простейшая программа. А где же предел? Какими минимальными знаниями и умениями должен обладать программист, чтобы утверждать, что он является, если конечно можно так выразиться, профессиональным Windows-программистом. Не претендую на безусловную истину, попытаемся перечислить некоторые проблемы, которые программист должен научиться решать в первую очередь:

- Понять общие принципы построения программы, работа которой управляется сообщениями.
- Научиться выводить текст и графику в область окна приложения. Основная проблема здесь состоит в умении эффективно использовать совокупность средств Win32 API для вывода результатов работы приложения в окно. Сам процесс формирования изображения в окне Windows напоминает процесс формирования изображения в видеобуфере, как это делалось в MS-DOS. Оба эти варианта вывода изображения можно сравнить с рисованием цветными мелками на школьной доске. Для того чтобы обновить содержимое окна, его не-

обходимо либо полностью вывести заново, либо сначала удалить ненужные фрагменты, сформировать на их месте новые и затем вывести их в определенное место в окне. Эта проблема называется проблемой *перерисовки изображения* и она тесно связана с тем, насколько эффективно решается следующая проблема.

- Организовать адекватную обработку сообщений. Эффективность и правильность работы программы напрямую зависит от того, насколько правильно в ней организована обработка сообщений. В самом начале процесса обучения написанию программ для Windows вы столкнетесь с необходимостью обработки такого сообщения, как `WM_PAINT`. Проблема здесь заключается в том, что Windows не сохраняет содержимое окна или части окна при его свертывании или сокрытии под другим окном. Следить за содержимым своих окон должно само приложение, а точнее, соответствующая оконная функция. Более подробно решение этой проблемы мы узнаем при рассмотрении вопроса перерисовки изображения.
- Научиться создавать интерфейсную часть приложения. Интерфейс приложения – это его визитная карточка. Первым, на что обращает внимание пользователь, тем более, если он непрофессионал, является именно этот элемент работы приложения. Более того, движущей силой развития самой Windows является стремление к реализации идеи идеального интерфейса. На сегодняшний день эту роль выполняет оконный интерфейс. Что будет завтра, пока не ясно, так как оконный интерфейс действительно решает многие проблемы и до исчерпания его полезных свойств, наверное, еще далеко. Основа оконного интерфейса – окно, в котором имеются две области: управляющая, с ее помощью осуществляется управление работой окна; пользовательская, которая обычно занимает большую часть окна, и именно в ней пользователь осуществляет формирование некоторого изображения. Управляющая область окна приложения состоит из более элементарных интерфейсных компонентов: меню, диалоговых окон, кнопок, панелей и т. д. Создание и организация работы со многими из этих компонентов поддерживается Windows с помощью функций Win32 API.
- Научиться обрабатывать пользовательский ввод.

Каждый пункт списка представляет лишь вершину некоторой иерархии более частных проблем и может быть достаточно глубоко детализирован вглубь. Конечно, в рамках нашего изложения этого сделать не представляется возможным, да и вряд ли нужно. Эти вопросы хорошо изложены в литературе. Материал данного урока подобран так, чтобы показать, как отражается на процессе разработки Windows-приложения выбор языка ассемблера в качестве основного языка программирования. Поэтому мы основное внимание уделяем не тому, как реализовать те или иные элементы пользовательского интерфейса, а технологии сборки работоспособного приложения с использованием средств пакета TASM. Исходя из этого, в следующей части нашего занятия покажем, как использовать ресурсы в Windows-приложениях, написанных на ассемблере.

Ресурсы Windows-приложений на языке ассемблера

Для включения ресурсов в Windows-приложения, написанные на ассемблере, используется та же самая технология, что и для программ на языке C/C++. Ресурс –

это специальный объект, используемый программой, но не определяемый в ее теле. К ресурсам относятся следующие элементы пользовательского интерфейса: значки, меню, окна диалога, растровые изображения.

Определение ресурсов производится в текстовом файле с расширением .rc, в котором для описания каждого типа ресурса используются специальные операторы. Подготовку этого файла можно вести двумя способами: ручным и автоматизированным.

Ручной способ предполагает следующее:

- разработчик ресурса хорошо знает операторы, необходимые для описания конкретного ресурса;
- ввод текста ресурсного файла выполняется с помощью редактора, который не вставляет в вводимый текст элементы форматирования. Наиболее доступный редактор для этих целей входит в состав программного обеспечения Windows — это `notepad.exe`.

Автоматический способ создания ресурсного файла предполагает использование специальной программы — редактора ресурсов, который позволяет визуализировать процесс создания ресурса. Конечные результаты работы этой программы могут быть двух видов: в виде текстового файла с расширением .rc, который впоследствии можно подвергнуть ручному редактированию, либо в виде двоичного файла, уже пригодного к включению в исполняемый файл приложения.

Будем предполагать, что мы одним из этих способов получаем файл ресурсов в текстовом виде.

После того как ресурсы описаны в файле с расширением .rc, они должны быть преобразованы в вид, пригодный для включения их в общий исполняемый файл приложения. Для этого необходимо:

1. Откомпилировать ресурсный файл. На этом шаге выполняется преобразование текстового представления ресурсного файла с расширением .rc в двоичное представление с расширением .res. Для реализации этого действия в пакете TASM есть специальная программа `brc32.exe` — компилятор ресурсов.
2. Включить ресурсы в исполняемый файл приложения. Это действие выполняет компоновщик `tlink32.exe`, которому в качестве последнего параметра должно быть передано имя ресурсного файла (.res).

Далее на конкретных примерах мы рассмотрим порядок применения этих программных средств для включения некоторых типов ресурсов в Windows-приложения.

Меню в Windows-приложениях

Меню в системе Windows являются, пожалуй, самым распространенным элементом пользовательского интерфейса. Мы не будем сильно вдаваться в детали разработки приложения с использованием меню, так как это уже делалось при рассмотрении каркасного приложения. Содержание этого процесса стандартное, поэтому мы остановимся только на специфических моментах его реализации при разработке приложения на ассемблере. Для того чтобы включить меню в приложение, необходимо реализовать следующую последовательность шагов:

○ разработать сценарий меню. Перед тем как приступить к процессу включения меню в конкретное приложение, разработаем его логическую схему. Этот шаг необходим для того, чтобы уже на стадии проектирования обеспечить эргономические свойства приложения. Ведь меню — это один из немногих элементов интерфейса, с которым пользователь вашего приложения будет постоянно иметь дело. Поэтому схема меню должна иметь наглядную иерархическую структуру, с логически связанными между собой пунктами этой иерархии, что поможет пользователю эффективно использовать все возможности приложения. Для того чтобы вести предметный разговор, поставим себе задачу разработать для окна нашего приложения главное меню. При этом мы исследуем возможности вывода в окно приложения текста и графики, а также покажем способы решения общих проблем, связанных с разработкой приложения. Наше меню достаточно простое и состоит из трех элементов: «Текст», «Графика» и «О приложении». Первые два из этих пунктов меню имеют вложенные всплывающие меню. Иерархическая структура меню представлена на рис. 18.2.



○ описать схему меню в файле ресурсов. Для выполнения этого описания используются специальные операторы. В нашем случае файл ресурсов будет выглядеть следующим образом:

```

//Текст файла menu.rc
#include "menu.h"
MYMENU MENU DISCARDABLE
{
POPUP "&Текст"
{
    MENUITEM "&DrawText", IDM_DRAWTEXT
    MENUITEM "&TextOut", IDM_TEXTOUT
}
POPUP "&Графика"
{
    POPUP "&Примитивы"
    {
        MENUITEM "&Отрезок", IDM_LENGTH
        MENUITEM "&Прямоугольник", IDM_RECTANGLE
    }
    POPUP "&Эффекты"
    {
        MENUITEM "&Павлин", IDM_PEACOCK
        MENUITEM "&Кружева", IDM_LACES
    }
}
MENUITEM "&О приложении", IDM_ABOUT
}
  
```

- составить текст включаемого файла, необходимого для компиляции ресурсного файла. В нашем случае файл называется `menu.h` и выглядит следующим образом:

```
#define    MYMENU 999
#define    IDM_DRAWTEXT 100
#define    IDM_TEXTOUT 101
#define    IDM_LENGTH 102
#define    IDM_RECTANGLE 103
#define    IDM_PEACOCK 104
#define    IDM_LACES 105
#define    IDM_ABOUT 106
```

В этом файле идентификаторам пунктов меню назначаются константы, которые впоследствии будут передаваться в оконную процедуру в младшем слове параметра `wParam` сообщения `WM_COMMAND`. Заметьте, что имени самого меню также назначена константа. К этому моменту мы вернемся чуть позже;

- скомпилировать ресурсный файл. Для этого используется утилита `brc32.exe`:
`brc32 [опции ...] menu.rc`

Если утилита заканчивает свою работу нормально, то создается файл с расширением `.res` (`menu.res`). В случае, если утилита обнаруживает ошибки в исходном ресурсном файле, то она выдает на экран соответствующие диагностические сообщения. Для удобства работы с ними их можно записать в файл, перенаправив вывод сообщений с экрана в файл, используя оператор командной строки `<>`:

```
brc32 [опции ...] имя_файла.rc > ИмяФайлаДляДиагностическихСообщений  
например, brc32 menu.rc > r
```

- подключить меню на стадии регистрации того окна приложения, для работы с которым оно будет использоваться. Для этого существуют два основных способа:

- поместить в поле `lpszMenuName` экземпляра структуры `WNDCLASS` (строка 101 листинга 18.4) указатель на поле, содержащее имя меню:

```
.data
...
menu    db    «MYMENU»
...
.code
...
    mov    dword ptr wcl.lpszMenuName, offset menu
...
```

- назначить ресурсу меню символьическую константу, которая расположена в поле имени оператора `MENU` файла ресурсов:

56 MENU DISCARDABLE :оператор из файла ресурсов

Чтобы ее использовать, необходимо вставить в файлы, из которых собирается приложение, следующие строки:

:оператор из файла ресурсов
MYMENU MENU DISCARDABLE

:оператор из menu.h

```
#define MYMENU 56  
;оператор из menu.inc  
MYMENU equ 56
```

;строка 101 из листинга 18.4 должна выглядеть так:
mov dword ptr wc1.lpszMenuName, MYMENU

или

```
mov dword ptr wc1.lpszMenuName, 999
```

В последнем случае мы фактически смоделировали известный вам по программированию на языке C/C++ для среды системы Windows макрос **MAKEINTRESOURCE** (см. приложение 11).

После внесения всех изменений в верхней части окна появится область меню. Далее необходимо в оконную функцию включить команды, которые будут являться реакцией на выбор пунктов этого меню. Эта информация передается в младшем слове поля **wParam** сообщения **WM_COMMAND**. В листинге 18.7 приведен измененный текст каркасного приложения, дополненный меню.

Листинг 18.7. Пример приложения с использованием меню

```
<1> ;Пример приложения для Win32 с использованием меню  
<2> .386  
<3> locals ;разрешает применение локальных меток в программе  
<4> .model flat, STDCALL ;модель памяти flat,  
<5> ;STDCALL – передача параметров в стиле С (справа налево),  
<6> ;вызываемая процедура чистит за собой стек  
<7> include windowA.inc ;включаемый файл с описаниями базовых структур  
 ;и констант Win32  
<8> include menu.inc ;включаемый файл с определением имен  
 ;идентификатор меню  
<9> ;Объявление внешними используемых в данной программе функций Win32 (ASCII):  
<10> extrn GetModuleHandleA:PROC  
<11> extrn GetVersionExA:PROC  
<12> extrn GetCommandLineA:PROC  
<13> extrn GetEnvironmentStringsA:PROC  
<14> extrn GetEnvironmentStringsA:PROC  
<15> extrn GetStartupInfoA:PROC  
<16> extrn LoadIconA:PROC  
<17> extrn LoadCursorA:PROC  
<18> extrn GetStockObject:PROC  
<19> extrn RegisterClassExA:PROC  
<20> extrn CreateWindowExA:PROC  
<21> extrn ShowWindow:PROC  
<22> extrn UpdateWindow:PROC  
<23> extrn GetMessageA:PROC  
<24> extrn TranslateMessage:PROC  
<25> extrn DispatchMessageA:PROC  
<26> extrn ExitProcess:PROC  
<27> extrn PostQuitMessage:PROC  
<28> extrn DefWindowProcA:PROC  
<29> extrn PlaySoundA:PROC
```

```

<30> extrn    ReleaseDC:PROC
<31> extrn    TextOutA:PROC
<32> extrn    GetDC:PROC
<33> extrn    BeginPaint:PROC
<34> extrn    EndPaint:PROC
<35> extrn    MessageBoxA:PROC
<36> extrn    DrawTextA:PROC
<37> extrn    GetClientRect:PROC
<38> ;объявление оконной функции объектом, видимым за пределами данного кода
<39> public     WindowProc
<40> .data
<41> hwnd        dd      0
<42> hInst       dd      0
<43> ;lpVersionInformation OSVERSIONINFO      <?>
<44> wcl         WNDCLASSEX      <?>
<45> message     MSG      <?>
<46> ps          PAINTSTRUCT     <?>
<47> lpRect      RECT     <?>
<48> szClassName db      'Приложение Win32', 0
<49> szTitleName db      'Каркасное приложение Win32 на ассемблере', 0
<50> MesWindow   db      'Привет! Ну как вам процесс разработки приложения
                           на ассемблере?'
```

<51> MesWindowLen = \$-MesWindow

<52> playFileCreate db 'create.wav', 0

<53> playFilePaint db 'paint.wav', 0

<54> playFileDestroy db 'destroy.wav', 0

<55> .code

<56> start proc near

<57> ;точка входа в программу:

<58> ;начало стартового кода

<59> ...

<60> ;строки 55–72 листинга 18.4

<61> ...

<62> ;конец стартового кода

<63> WinMain:

<64> ...

<65> ;строки 75–100 листинга 18.4

<66> ...

<67> mov dword ptr wcl.lpszMenuName, MYMENU

<68> ;строки 102–149 листинга 18.4

<69> ...

<70> start endp

<71> -----WindowProc-----

<72> WindowProc proc

<73> arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD

<74> uses ebx, edi, esi, ebx ;эти регистры обязательно должны сохраняться

<75> local @@hdc:DWORD

<76> cmp @@mes, WM_DESTROY

<77> je wnddestroy

<78> cmp @@mes, WM_CREATE

```
<79>         je      wmccreate
<80>         cmp     @@mes, WM_PAINT
<81>         je      wmpaint
<82>         cmp     @@mes, WM_COMMAND
<83>         je      wmcommand
<84>         jmp     default
<85> wmccreate:
<86> ...
<87> ;строки 176–212 листинга 18.4
<88> ...
<89> ;послать сообщение WM_QUIT
<90> ;готовим вызов VOID PostQuitMessage(int nExitCode)
<91>         push    0
<92>         call    PostQuitMessage
<93>         mov     eax, 0 ;возвращаемое значение – 0
<94>         jmp     exit_wndproc
<95> wmcommand:
<96> ;вызов процедуры обработки сообщений от меню
<97> ;MenuProc (DWORD @@hwnd, DWORD @@@wparam)
<98>         push    @@@wparam
<99>         push    @@hwnd
<100>        call   MenuProc
<101>        jmp     exit_wndproc
<102> default:
<103> ;обработка по умолчанию
<104> ;готовим вызов LRESULT DefWindowProc(HWND hWnd, UINT Msg, WPARAM wParam,
                                         LPARAM lParam)
<105>         push    @@@lparam
<106>         push    @@@wparam
<107>         push    @@mes
<108>         push    @@hwnd
<109>         call   DefWindowProcA
<110>         jmp     exit_wndproc
<111> ;.....
<112> exit_wndproc:
<113>         ret
<114> WindowProc endp
<115> ;-----MenuProc-----
<116> ;обработка сообщений от меню
<117> MenuProc proc
<118> arg      @@hwnd:DWORD, @@@wparam:DWORD
<119> uses    ebx
<120> local   @@hdc:DWORD
<121>         mov     ebx, @@@wparam ;в bx идентификатор меню
<122>         cmp     bx, IDM_DRAWTEXT
<123>         je      @@idmdrawtext
<124>         cmp     bx, IDM_TEXTOUT
<125>         je      @@idmtextout
<126>         cmp     bx, IDM_LENGTH
<127>         je      @@idmlength
<128>         cmp     bx, IDM_RECTANGLE
<129>         je      @@idmrectangle
```

Листинг 18.7 (продолжение)

```
<130>     cmp      bx, IDM_PEACOCK
<131>     je       @@idmpeacock
<132>     cmp      bx, IDM_LACES
<133>     je       @@idmlaces
<134>     cmp      bx, IDM_ABOUT
<135>     je       @@idmabout
<136>     jmp      @@exit
<137> @@idmdrawtext:
<138> ;получим контекст устройства HDC GetDC(HWND hWnd)
<139>     push    @@hwnd
<140>     call    GetDC
<141>     mov     @@hdc, eax
<142> ;получим размер рабочей области BOOL GetClientRect(HWND hWnd, LPRECT lpRect)
<143>     push    offset lpRect
<144>     push    @@hwnd
<145>     call    GetClientRect
<146> ;выведем строку текста в окно int DrawText(HDC hdc, LPCTSTR lpString, int nCount,
<147> :          LPRECT lpRect, UINT uFormat)
<148>     push    DT_SINGLELINE+DT_BOTTOM
<149>     push    offset lpRect
<150>     push    -1
<151>     push    offset @@TXT_DRAWTEXT
<152>     push    @@hdc
<153>     call    DrawTextA
<154> ;освободить контекст int ReleaseDC(HWND hWnd, HDC hdc)
<155>     push    @@hdc
<156>     push    @@hwnd
<157>     call    ReleaseDC
<158>     jmp    @@exit
<159> @@idmtextout:
<160>     push    @@hwnd
<161>     call    GetDC ;получим контекст устройства
<162>     mov     @@hdc, eax
<163> ;выведем строку текста в окно BOOL TextOut(HDC hdc, int nXStart, int
<164> :          nYStart,
<165>     push    lenTXT_TEXTOUT
<166>     push    offset @@TXT_TEXTOUT
<167>     push    150
<168>     push    10
<169>     push    @@hdc
<170>     call    TextOutA
<171>     push    @@hdc
<172>     push    @@hwnd
<173>     call    ReleaseDC ;освободим контекст устройства
<174>     jmp    @@exit
<175> @@idmlength:
<176>     push    MB_ICONINFORMATION+MB_OK
<177>     push    offset szTitleName
```

```

<178>      push    offset @@TXT_LENGTH
<179>      push    @@hwnd
<180>      call    MessageBoxA
<181>      jmp    @@exit
<182> @@idmrectangle:
<183>      push    MB_ICONINFORMATION+MB_OK
<184>      push    offset szTitleName
<185>      push    offset @@TXT_RECTANGLE
<186>      push    @@hwnd
<187>      call    MessageBoxA
<188>      jmp    @@exit
<189> @@idmpeacock:
<190>      push    MB_ICONINFORMATION+MB_OK
<191>      push    offset szTitleName
<192>      push    offset @@TXT_PEACOCK
<193>      push    @@hwnd
<194>      call    MessageBoxA
<195>      jmp    @@exit
<196> @@idmlaces:
<197>      push    MB_ICONINFORMATION+MB_OK
<198>      push    offset szTitleName
<199>      push    offset @@TXT_LACES
<200>      push    @@hwnd
<201>      call    MessageBoxA
<202>      jmp    @@exit
<203> @@idmabout:
<204>      push    MB_ICONINFORMATION+MB_OK
<205>      push    offset szTitleName
<206>      push    offset @@TXT_ABOUT
<207>      push    @@hwnd
<208>      call    MessageBoxA
<209>      jmp    @@exit
<210> .....
<211> @@exit:
<212>      mov     eax, 0
<213>      ret
<214> @@TXT_ABOUT db   'IDM_ABOUT', 0
<215> @@TXT_LACES db   'IDM_LACES', 0
<216> @@TXT_PEACOCK db   'IDM_PEACOCK', 0
<217> @@TXT_RECTANGLE db   'IDM_RECTANGLE', 0
<218> @@TXT_LENGTH db   'IDM_LENGTH', 0
<219> @@TXT_TEXTOUT db   'Текст выведен функцией TEXTOUT'
<220> 1enTXT_TEXTOUT=$-@@TXT_TEXTOUT
<221> @@TXT_DRAWTEXT db   'Текст выведен функцией DRAWTEXT', 0
<222> MenuProc endp
<223> end      start

```

Строки 98–100 листинга 18.7 показывают, что для обработки сообщения WM_COMMAND вызывается процедура MenuProc. Подход, при котором программа на ассемблере структурируется на более мелкие части с использованием механизма процедур, при программировании для Windows особенно полезен и должен быть преобладающим. Для большей наглядности изложения материала мы не использу-

зум макрокоманды. Применение макрокоманд полезно и даже необходимо, так как они позволяют получать компактный и структурированный код.

В заключение этого раздела обратитесь к содержимому дискеты, где в каталоге для данного урока вы найдете исходные тексты всех файлов, необходимые для сборки и запуска приложения (см. листинг 18.7). Собрав с помощью утилиты `make.exe` исполняемый файл, вы увидите, что реакция программы на выбор большинства пунктов меню стандартная и заключается в выводе сообщения с использованием функции Win32 API `MessageBox`. Определена реакция лишь на два пункта выпадающего меню «Текст» — пункты «DrawText» и «TextOut». При их выборе на экран выводится текст. Тем самым демонстрируются различные способы вывода текста в окно приложения. Далее в материале этого и последующих уроков мы определим реакцию и на остальные пункты меню.

Давайте теперь рассмотрим еще одну ключевую проблему программирования для Windows — *перерисовку изображения*. Для того чтобы понять ее содержание и важность, выполните следующую последовательность действий. В любом порядке выберите пункты меню Текст ▶ DrawText и Текст ▶ TextOut. На экране появятся строки текста. Далее сверните и разверните окно приложения. Строки текста исчезли, осталась только строка, которая выводится как реакция на сообщение `WM_PAINT`. Причина этой ситуации в том, что Windows не хранит содержимое окна, и заботиться о его восстановлении после различных с ним действий должно приложение, создавшее это окно. Windows лишь посыпает приложению сообщение `WM_PAINT` в случаях, когда с окном были произведены некоторые действия, например, свертывание/развертывание окна, изменение его размеров и т. д. Приложение, получив сообщение `WM_PAINT`, в качестве реакции на него должно обновить в необходимой степени содержимое своего окна. Более подробно эта проблема и пути ее решения рассмотрены в любом хорошем учебнике по программированию для системы Windows. Отметим лишь, что в конечном итоге все сводится к необходимости организации вывода в окно приложения как реакции на сообщение `WM_PAINT`. Существует несколько подходов к решению проблемы перерисовки. Наиболее общий и широко используемый способ перерисовки изображения основан на понятии *виртуального окна*. Рассмотрим его.

Перерисовка изображения

Рассмотрим следующую абстракцию, цель которой — показать, что за содержимое окон на экране отвечают оконные процедуры тех приложений, которым эти окна принадлежат. Роль Windows в этом процессе минимальна и состоит в том, что при определенных действиях с окном оконной процедуре, отвечающей за связь с этим окном, посыпается сообщение `WM_PAINT`. Получив это сообщение, оконная процедура должна уметь заново перерисовать содержимое всего окна или его части. Для общего случая это, если задуматься, не такая уж простая задача.

Если у вас возникают трудности с пониманием этого момента работы приложений в Windows, то попробуйте развить приведенную выше абстракцию со школьной доской. Представьте себе школьный класс, в котором процесс обучения выглядит очень необычно. Вместо обычной школьной доски на стене есть некоторая ограниченная область (экран монитора). У каждого ученика (исполняемого файла) есть в портфеле своя доска (окно приложения), которая может (или не мо-

жет, в зависимости от конструкции) менять свой размер. На всех учеников один комплект цветных мелков (контекст устройства), обладание которым разрешает ученику что-то рисовать на своей доске. Ученик объясняется с присутствующими графическими образами, выводя их на свою доску. Но прежде доску нужно повесить на стену в пределах ограниченной области. Для этого ученик, как ему и положено, подымает руку. Учитель (Windows) обязательно помогает ученику выйти к области на стене (запускает на выполнение задачу) и повесить свою доску. Действия учителя и ученика в ходе этого процесса соответствуют определенному алгоритму и действиям каркасного приложения. В результате могут быть закрыты частично или полностью доски учеников, вышедших к доске ранее и не ушедших пока на свои места в классе. При этом информация в скрытых частях нижерасположенных досок других учеников уничтожается.

Итак, доска ученика с помощью учителя повешена в пределах области на стене, и для того, чтобы рисовать на ней, ученик должен попросить у учителя комплект мелков (см. в листинге 18.4 функции API `GetDC` или `BeginPaint`). Если ученик его получает, то он может начинать процесс рисования на своей доске. Как только ученик закончил этот процесс, он должен вернуть учителю комплект мелков (функция `EndPaint`). Далее ученик может закончить ответ и, забрав свою доску, занять свое место в классе (приложение завершило работу и «ушло» на диск). Если ученик сделал это, то забрав свою доску, он открыл доски других учеников, и, о, какой позор — на этих досках зияют черные дыры. Местоположение этих дыр соответствуют местам, которые были скрыты доской их ушедшего товарища. Чтобы сгладить этот конфуз, учитель срочно оповещает об этом (Windows посылает сообщение `WM_PAINT`) учеников, у которых было испорчено содержимое досок. Каждый из этих учеников должен, попросив у учителя комплект мелков, перерисовать содержимое своей доски. Данная абстракция достаточно точно отражает логику работы Windows, причем не только в обозначенном нами контексте. Вы можете дополнить и развить ее в нужную вам сторону.

Как решается проблема перерисовки изображения практически? Для этого существует несколько способов, самый общий из которых заключается в использовании *виртуального окна*. Суть перерисовки изображения на основе виртуального окна заключается в использовании приложением некоторой области памяти для направления в него всего вывода программы. Реальный вывод в окно приложения осуществляется только как реакция на получение сообщения `WM_PAINT`. Не забывайте, что программа, используя функцию `InvalidateRect()`, может сама себе послать сообщение `WM_PAINT`. Приложение делает это, когда ему необходимо вывести новую порцию изображения в окно приложения. Понятие виртуального окна настолько важно для организации работы Windows, что Win32 API содержит ряд функций, поддерживающих работу с этим окном: `CreateCompatibleDC()`, `SelectObject()`, `GetStockObject()`, `BitBlt()`, `CreateCompatibleBitmap()` и `PatBlt()`. Посмотрим порядок их использования в реальном приложении.

Работа с виртуальным окном в программе организуется в два этапа: создание виртуального окна и организация непосредственной работы с ним. Создать виртуальное окно целесообразно при обработке сообщения `WM_CREATE`, то есть в момент создания окна приложения. Работать с этим окном можно в любое время, когда требуется выполнить вывод в окно.

Для наглядности обсуждения приведем фрагмент программы на языке С/C++:

//фрагмент оконной процедуры из программы на языке С/C++

...
case WM_CREATE:

//определить размеры экрана

maxX = GetSystemMetrics (SM_CXSCREEN);

maxY = GetSystemMetrics (SM_CXSCREEN);

// строим растровое изображение, совместимое с окном

hdc = GetDC (hwnd);

memdc = CreateCompatibleDC (hdc);

hbit = CreateCompatibleBitmap (hdc, maxX, maxY);

SelectObject (memdc, hbit);

//закрашиваем окно серым цветом

hbrush = GetStockObject (GRAY_BRUSH);

SelectObject (memdc, hbrush);

PatBlt (memdc, 0, 0, maxX, maxY, PATCOPY);

ReleaseDC (hwnd, hdc);

...
case WM_PAINT: // перерисовать окно

hdc = BeginPaint (hwnd, &paintstruct); // получить дескриптор реального окна

// копируем растровое изображение из памяти на экран

BitBlt (hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);

EndPaint (hwnd, &paintstruct); // освободить дескриптор

...
//вывод в окно где-то в программе

TextOut(memdc, X, Y, str, strlen(str)); //вывести строку

//если нужно немедленно произвести вывод в окно, то вызов функции InvalidateRect:

InvalidateRect (hwnd, NULL, 1);

Физически виртуальное окно представляет собой растровое изображение, хранящееся в памяти. Работа с этой областью памяти организуется так же, как и с окном приложения на экране монитора. Это означает, что для работы с ним необходимо создать контекст устройства памяти, совместимый с контекстом окна. Это действие реализуется двумя функциями: GetDC(), с помощью которой приложение получает контекст окна, и CreateCompatibleDC(), которая создает совместимый с контекстом окна контекст памяти memdc. После этого функцией CreateCompatibleBitmap() создается совместимое с реальным окном на экране растровое изображение. Его размеры должны соответствовать размеру окна, для работы с которым строится растровое изображение. Поэтому предварительно с помощью функций GetSystemMetrics() должны быть получены размеры окна и переданы как параметры в CreateCompatibleBitmap(). Функция CreateCompatibleBitmap() возвращает дескриптор на созданное растровое изображение. После этого функция SelectObject() выбирает созданное растровое изображение в контекст памяти, который, в свою очередь, совместим с контекстом окна. Благодаря такой цепочке связей обращение к растровому изображению в памяти производится аналогично обращению к реальному окну. На практике это означает, что во всех функциях, выводящих изображение в окно, на месте параметра, соответствующего контексту устройства, необходимо указывать контекст устройства памяти. Например, функция TextOut() будет вызываться следующим образом:

push lenTXT_TEXTOUT

push offset @@TXT_TEXTOUT

```
push    150
push    10
push    @memdc
call    TextOutA
```

В пятой строке этого фрагмента функции `TextOut()` передается не контекст окна, а контекст виртуального окна, что и приводит к выводу не в реальное окно, а в виртуальное, являющееся растровым изображением. Как мы уже отметили, в программе есть только одно место, где производится вывод в реальное окно, — это фрагмент программы, обрабатывающий сообщение `WM_PAINT`. В случае использования виртуального окна здесь располагается функция `BitBlt()`, которая копирует содержимое растрового изображения из контекста памяти в контекст реального окна. Таким образом, будет постоянно обеспечиваться актуальное содержимое окна приложения. В листинге 18.8 приведены фрагменты текста программы на языке ассемблера, демонстрирующей практическую реализацию способа перерисовки окна приложения с использованием виртуального окна. Полный текст программы приведен на диске в подкаталоге ..\prg19_3 данного урока.

Листинг 18.8. Фрагменты приложения prg19_3.asm

```
<1> ;Фрагменты приложения (prg19_3.asm) для Win32 с использованием меню
      и виртуального окна для перерисовки содержимого окна;
<2> .386
<3> locals   ;разрешает применение локальных меток (с префиксом @@) в программе
<4> .model flat, STDCALL ;модель памяти flat,
<5> ;STDCALL – передача параметров в стиле C (справа налево),
<6> ; вызываемая процедура чистит за собой стек
<7> include windowA.inc   ;включаемый файл с описаниями базовых структур
                           и констант Win32
<8> include      menu.inc ;включаемый файл с определением имен идентификатора
                           меню
<9> ;Объявление внешними используемых в данной программе функций Win32 (ASCII):
<10> extrn     GetModuleHandleA:PROC
<11> ... ...
<12> extrn     GetDC:PROC
<13> extrn     BeginPaint:PROC
<14> extrn     EndPaint:PROC
<15> extrn     MessageBoxA:PROC
<16> extrn     DrawTextA:PROC
<17> extrn     GetClientRect:PROC
<18> extrn     GetSystemMetrics:PROC
<19> extrn     CreateCompatibleDC:PROC
<20> extrn     CreateCompatibleBitmap:PROC
<21> extrn     SelectObject:PROC
<22> extrn     GetStockObject:PROC
<23> extrn     PatBlt:PROC
<24> extrn     BitBlt:PROC
<25> extrn     InvalidateRect:PROC
<26> extrn     DeleteDC:PROC
<27> ... ...
<28> .data
<29> memdc dd 0           !!!это глобальная переменная
```

Листинг 18.8 (продолжение)

```
<30> maxX      dd      0      ;!!!это глобальная переменная
<31> maxY      dd      0      ;!!!это глобальная переменная
<32> ...
<33> lpRect     RECT    <?>
<34> ...
<35> .code
<36> start      proc    near
<37> ;точка входа в программу:
<38> ...
<39> WinMain:
<40> ...
<41> start      endp
<42> ;-----WindowProc-----
<43> WindowProc  proc
<44> arg        @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
<45> uses ebx, edi, esi, ebx      ;эти регистры обязательно должны сохраняться
<46> local       @@hdc:DWORD, @@hbrush:DWORD, @@hbit:DWORD
<47>             cmp      @@mes, WM_DESTROY
<48>             je       wmdestroy
<49>             cmp      @@mes, WM_CREATE
<50>             je       wmccreate
<51>             cmp      @@mes, WM_PAINT
<52>             je       wmpaint
<53>             cmp      @@mes, WM_COMMAND
<54>             je       wmccommand
<55>             jmp      default
<56> wmccreate:
<57> ;создание растрового изображения, совместимого с окном приложения
<58> ;получим размер экрана в пикселях int GetSystemMetrics(int nIndex)
<59>             push    SM_CXSCREEN
<60>             call    GetSystemMetrics
<61>             mov     maxX, eax
<62>             push    SM_CYSCREEN
<63>             call    GetSystemMetrics
<64>             mov     maxY, eax
<65> ;получить контекст устройства окна на экране @@hdc=GetDC(@@hwnd)
<66>             push    @@hwnd
<67>             call    GetDC
<68>             mov     @@hdc, eax
<69> ;получить совместимый контекст устройства памяти
<70> ;memdc=CreateCompatibleDC(@@hdc)
<71>             push    @@hdc
<72>             call    CreateCompatibleDC
<73>             mov     memdc, eax  ;!! memdc – глобальная переменная
<74> ;получить дескриптор растрового изображения в памяти
<75> ; @@hbit=CreateCompatibleBitmap(@@hdc, @@maxX, @@maxY)
<76>             push    maxY
<77>             push    maxX
<78>             push    @@hdc
```

```
<79>           call    CreateCompatibleBitmap
<80>           mov     @@hbit, eax
<81> ; выбираем растр в контекст памяти SelectObject(memdc, @@hbit)
<82>           push    @@hbit
<83>           push    memdc
<84>           call    SelectObject
<85> ; выполним первичное заполнение растра серым цветом
<86> ; получим дескриптор серой кисти hbrush=GetStockObject(GRAY_BRUSH)
<87>           push    GRAY_BRUSH
<88>           call    GetStockObject
<89>           mov     @@hbrush, eax
<90> ; выбираем кисть в контекст памяти SelectObject(memdc, @@hbrush)
<91>           push    @@hbrush
<92>           push    memdc
<93>           call    SelectObject
<94> ; заполняем выбранной кистью виртуальное окно
<95> ; BOOL PatBlt(HDC hdc, int nXLeft, int nYLeft, int nWidth, int nHeight,
                  DWORD dwRop)
<96>           push    PATCOPY
<97>           push    maxY
<98>           push    maxX
<99>           push    NULL
<100>          push    NULL
<101>          push    memdc
<102>          call    PatBlt
<103> ; освободим контекст устройства ReleaseDC(@@hwnd, @@hdc)
<104>          push    @@hdc
<105>          push    @@hwnd
<106>          call    ReleaseDC
<107> ; обозначим создание окна звуковым эффектом
<108> ; готовим вызов функции BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD
                  fdwSound)
<109>          push    SND_SYNC+SND_FILENAME
<110>          push    NULL
<111>          push    offset playFileCreate
<112>          call    PlaySoundA
<113> ; возвращаем значение 0
<114>          mov     eax, 0
<115>          jmp    exit_wndproc
<116> wmpaint:
<117> ; получим контекст устройства HDC BeginPaint(HWND hwnd, LPPAINTSTRUCT
                  lpPaint);
<118>          push    offset ps
<119>          push    @@hwnd
<120>          call    BeginPaint
<121>          mov     @@hdc, eax
<122> ; обозначим перерисовку окна звуковым эффектом
<123>          push    SND_SYNC+SND_FILENAME
<124>          push    NULL
<125>          push    offset playFilePaint
<126>          call    PlaySoundA
```

Листинг 18.8 (продолжение)

```
<127> ;выведем строку текста в окно BOOL TextOut(HDC hdc, int nXStart, int
      nYStart,
<128> :           LPCTSTR lpString, int cbString);
<129>     push    MesWindowLen
<130>     push    offset MesWindow
<131>     push    100
<132>     push    10
<133>     push    memdc
<134>     call    TextOutA
<135> ;вывод виртуального окна в реальное окно
<136> ;BOOL BitBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth, int nHeight,
<137> :           HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwRop)
<138>     push    SRCCOPY
<139>     push    NULL
<140>     push    NULL
<141>     push    memdc
<142>     push    maxY
<143>     push    maxX
<144>     push    NULL
<145>     push    NULL
<146>     push    @@hdc
<147>     call    BitBlt
<148> ;освободить контекст BOOL EndPaint(HWND hWnd, CONST PAINTSTRUCT *lpPaint);
<149>     push    offset ps
<150>     push    @@hWnd
<151>     call    EndPaint
<152>     mov     eax, 0 ;возвращаемое значение – 0
<153>     jmp    exit_wndproc
<154> wmdestroy:
<155> ... ...
<156> wmcancel:
<157> ;вызов процедуры обработки сообщений от меню
<158> ;MenuProc (DWORD @@hWnd, DWORD @@@wParam)
<159>     push    @@@wParam
<160>     push    @@hWnd
<161>     call    MenuProc
<162>     jmp    exit_wndproc
<163> default:
<164> ;.... ....
<165> exit_wndproc:
<166>     ret
<167> WindowProc endp
<168> ;-----MenuProc-----
<169> ;обработка сообщений от меню
<170> MenuProc proc
<171> arg    @@hWnd:DWORD, @@@wParam:DWORD
<172> uses   ebx
<173> local
<174>     mov     ebx, @@@wParam ;в bx идентификатор меню
<175> ... ...
```

```
<176>     jmp      @@exit
<177> @@idmdrawtext:
<178> ;получим размер рабочей области BOOL GetClientRect(HWND hWnd, LPRECT
                                         lpRect);
<179>         push    offset lpRect
<180>         push    @@hwnd
<181>         call    GetClientRect
<182> ;выведем строку текста в окно int DrawText(HDC hdc, LPCTSTR lpString, int
                                         nCount, length,
<183> ;           LPRECT lpRect, UINT uFormat);
<184>         push    DT_SINGLELINE+DT_BOTTOM
<185>         push    offset lpRect
<186>         push    -1
<187>         push    offset @@TXT_DRAWTEXT
<188>         push    memdc
<189>         call    DrawTextA
<190> ;генерация сообщения WM_PAINT для вывода строки на экран
<191> ;BOOL InvalidateRect(HWND hWnd, CONST RECT *lpRect, BOOL bErase)
<192>         push    1
<193>         push    NULL
<194>         push    @@hwnd
<195>         call    InvalidateRect
<196>         jmp    @@exit
<197> @@idmtextout:
<198> ;выведем строку текста в окно BOOL TextOut(HDC hdc, int nXStart,
<199> ;           int nYStart, LPCTSTR lpString, int cbString)
<200>         push    lenTXT_TEXTOUT
<201>         push    offset @@TXT_TEXTOUT
<202>         push    150
<203>         push    10
<204>         push    memdc
<205>         call    TextOutA
<206> ;генерация сообщения WM_PAINT для вывода строки на экран
<207>         push    0
<208>         push    NULL
<209>         push    @@hwnd
<210>         call    InvalidateRect
<211>         jmp    @@exit
<212> @@idmlength:
<213> ... ...
<214>         jmp    @@exit
<215> @@idmrectangle:
<216> ... ...
<217>         jmp    @@exit
<218> @@idmpeacock:
<219> ... ...
<220>         jmp    @@exit
<221> @@idmlaces:
<222> ... ...
<223>         jmp    @@exit
<224> @@idmabout:
<225> ... ...
```

```

<226>           jmp    @@exit
<227> ;.....
<228> @@exit:
<229>           mov    eax, 0
<230>           ret
<231> @@TXT_ABOUT db     'IDM_ABOUT', 0
<232> @@TXT_LACES db    'IDM_LACES', 0
<233> @@TXT_PEACOCK db   'IDM_PEACOCK', 0
<234> @@TXT_RECTANGLE db  'IDM_RECTANGLE', 0
<235> @@TXT_LENGTH db   'IDM_LENGTH', 0
<236> @@TXT_TEXTOUT db   'Текст выведен функцией TEXTOUT'
<237> lenTXT_TEXTOUT=$-@@TXT_TEXTOUT
<238> @@TXT_DRAWTEXT db   'Текст выведен функцией DRAWTEXT', 0
<239> MenuProc endp
<240> end      start

```

Возможно, результаты работы программы листинга 18.8 вам покажутся не очень красивыми, но такая цель и не ставилась. Цели этой программы — исследовательские. Из-за задержек, вызванных воспроизведением звуковых файлов, хорошо становится виден момент перерисовки окна. Такую технологию можно использовать для более глубокого исследования механизмов работы Windows.

Окна диалога в Windows-приложениях

Окна диалога являются важными и широко используемыми элементами пользовательского интерфейса, предоставляемого системой Windows. Редкое оконное приложение обходится без использования окон этого типа. Физически окно диалога представляет собой специфическое окно, работа с которым поддерживается на уровне интерфейса Win32 API Windows. Основное назначение этого окна — помочь пользователю сформировать информацию, необходимую для управления работой приложения. Наиболее наглядные примеры окон этого типа — это окна диалога, используемые в текстовом редакторе. С их помощью вы задаете параметры шрифта или параметры, необходимые для печати документа. Очень важно, что разработка таких окон не требует программирования. Для описания окон диалога система Windows поддерживает специальный тип ресурса. Более подробно о том, как в программе должны выглядеть окна диалога, из каких элементов они состоят и о деталях управления этими окнами вы можете почитать в других источниках. Нашей целью, как обычно, является показать то, каким образом работать с окнами диалога организуется программой, написанной на языке ассемблера. С точки зрения технологии, организация работы с окнами диалога программой на языке ассемблера ничем не отличается от того, как это делается программой на любом другом языке. Чтобы создать окно диалога на языке ассемблера необходимо выполнить следующие шаги:

- описать окно диалога в файле ресурсов;
- разработать *диалоговую функцию*. Данная функция будет обрабатывать сообщения, предназначенные для определенного в файле ресурсов окна диалога;
- активизировать диалог в приложении.

Для того чтобы разговор был предметным, поставим себе цель разработать конкретную программу. В последней программе (см. листинг 18.8) мы предусмотрели возможность ее расширения. Сейчас для этого настало время. Дополним программу листинга 18.8, фрагментами, обеспечивающими работу пункта меню **Графика** ► **Примитивы** (см. рис. 18.2). Это меню имеет раскрывающееся подменю, пункты которого называются «Отрезок» и «Прямоугольник». Обеспечим работу данных пунктов меню с помощью окон диалога, задача которых будет заключаться в принятии от пользователя параметров простейших фигур — отрезка и прямоугольника и отрисовки их в окне приложения в соответствии с заданными параметрами. Параллельно мы разберемся с некоторыми общими принципами работы с графикой, понимая которые, вы сможете реализовать более сложные алгоритмы. Текст приложения приведен в листинге 18.13.

Описание окна диалога в файле ресурсов

Как было отмечено, окно диалога представляет собой специальный объект, предназначенный для облегчения взаимодействия пользователя с выполняющейся программой и настройки ее на определенные условия функционирования. Окно диалога состоит из элементарных объектов, называемых *элементами управления*. Система Windows поддерживает несколько типов таких объектов. В рассматриваемом далее примере мы будем использовать только небольшую их часть.

Для описания внешнего вида окна диалога и обеспечения его интерфейса с приложением используется специальный тип ресурса — **DIALOG**. В отличие от ресурса меню, который можно создать вручную, ресурс диалогового окна лучше создавать с помощью соответствующих программных средств — редакторов ресурсов. Основная причина здесь в том, что при ручном определении размеров и взаимного расположения элементов управления трудно представить то, как это будет выглядеть на экране. Сформулируем некоторые общие принципы, которые позволяют вам без труда выполнять эту часть работы.

TASM 5.0 не имеет своего редактора ресурсов, поэтому вам придется поискать его на стороне. В качестве такого редактора не обязательно должен быть автономный редактор ресурсов — вполне подойдет редактор, встроенный в интегрированную среду разработки для языка высокого уровня. Главное требование, чтобы он создавал файл описания ресурса с расширением **.rc**. В файле описания, наряду со строками, описывающими ресурс, будут строки, необходимые для обеспечения работы интегрированной среды с ресурсами. Их нужно просто удалить. Строки, содержащие непосредственное описание ресурса, включите в общий файл описания ресурсов приложения.

Ресурсы диалоговых окон описанного далее приложения создавались с помощью редактора ресурсов VC ++ версии 4.2. Ресурс каждого диалогового окна создавался отдельно, а затем добавлялся в файл ресурсов приложения **prg18.4.rc**. В этом же файле, кстати, уже находился ранее разработанный ресурс с описанием меню приложения. Параллельно с созданием файла ресурсов приложения необходимо создавать (или модифицировать) включаемый файл описаний, содержащий символьные константы элементов меню и окон диалога. Для нашего примера файл ресурсов имеет описание, представленное в листинге 18.9.

Листинг 18.9. Файл ресурсов prg19_4.rc

```
<1> #include      "Prg19_4.h"
<2> #include      <windows.h>
<3> IDI_ICON1    ICON      DISCARDABLE "icon1.ico"
<4> MYMENU MENU DISCARDABLE
<5> {
<6> POPUP "&Текст"
<7> {
<8>   MENUITEM "&DrawText", IDM_DRAWTEXT
<9>   MENUITEM "&TextOut", IDM_TEXTOUT
<10>  }
<11> POPUP "&Графика"
<12> {
<13>   POPUP "&Примитивы"
<14>   {
<15>     MENUITEM "&Отрезок", IDM_LENGTH
<16>     MENUITEM "&Прямоугольник", IDM_RECTANGLE
<17>   }
<18>   POPUP "&Эффекты"
<19>   {
<20>     MENUITEM "&Павлин", IDM_PEACOCK
<21>     MENUITEM "&Кружева", IDM_LACES
<22>   }
<23> }
<24> MENUITEM "&About", IDM_ABOUT
<25> }
<26> ///////////////////////////////////////////////////////////////////
<27> //
<28> // Dialog для отрезка
<29> //
<30> IDD_DIALOG1 DIALOG DISCARDABLE 0, 0, 186, 95
<31> STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
<32> CAPTION "Отрезок"
<33> FONT 8, "MS Sans Serif"
<34> BEGIN
<35>   DEFPUSHBUTTON "OK", IDOK, 35, 72, 50, 14
<36>   PUSHBUTTON "Cancel", IDCANCEL, 118, 72, 50, 14
<37>   LTEXT "Задайте координаты концов отрезка:", IDC_STATIC, 22,
       6, 134, 8
<38>   EDITTEXT IDC_EDIT1, 34, 37, 20, 12, ES_AUTOHSCROLL
<39>   LTEXT "Xstart", IDC_STATIC, 5, 40, 19, 8
<40>   LTEXT "Ystart", IDC_STATIC, 5, 54, 19, 8
<41>   LTEXT "Xend", IDC_STATIC, 91, 39, 18, 8
<42>   LTEXT "Yend", IDC_STATIC, 91, 52, 18, 8
<43>   EDITTEXT IDC_EDIT2, 34, 52, 20, 12, ES_AUTOHSCROLL
<44>   EDITTEXT IDC_EDIT3, 118, 36, 20, 12, ES_AUTOHSCROLL
<45>   EDITTEXT IDC_EDIT4, 118, 52, 20, 12, ES_AUTOHSCROLL
<46> END
<47> ///////////////////////////////////////////////////////////////////
<48> //
<49> // Dialog для прямоугольника
```

```

<50> // 
<51> IDD_DIALOG2 DIALOG DISCARDABLE      0, 0, 186, 95
<52>   STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
<53>   CAPTION "Прямоугольник"
<54>   FONT 8, "MS Sans Serif"
<55> BEGIN
<56>       DEFPUSHBUTTON    "OK", IDOK, 35, 72, 50, 14
<57>       PUSHBUTTON      "Cancel", IDCANCEL, 118, 72, 50, 14
<58>       LTEXT            "Задайте координаты углов прямоугольника:", IDC_STATIC, 22, 6, 174, 8
<59>       EDITTEXT        IDC_EDIT1, 34, 37, 20, 12, ES_AUTOHSCROLL
<60>       LTEXT            "X", IDC_STATIC, 22, 39, 8, 8
<61>       LTEXT            "Y", IDC_STATIC, 21, 55, 8, 8
<62>       LTEXT            "X", IDC_STATIC, 104, 39, 8, 8
<63>       LTEXT            "Y", IDC_STATIC, 104, 54, 8, 8
<64>       EDITTEXT        IDC_EDIT2, 34, 52, 20, 12, ES_AUTOHSCROLL
<65>       EDITTEXT        IDC_EDIT3, 118, 36, 20, 12, ES_AUTOHSCROLL
<66>       EDITTEXT        IDC_EDIT4, 118, 52, 20, 12, ES_AUTOHSCROLL
<67>       LTEXT            "Left_Top:", IDC_STATIC, 4, 27, 32, 8
<68>       LTEXT            "Right_Bottom:", IDC_STATIC, 65, 27, 46, 8
<69> END
<70> ///////////////////////////////////////////////////////////////////
<71> //
<72> // Dialog для пункта меню About
<73> //
<74> AboutBox DIALOG      20, 20, 160, 80
<75>   STYLE WS_POPUP | WS_DLGFRADE
<76> {
<77>     CTEXT            "TASM32"      -1, 0, 12, 160, 8
<78>     ICON              "IDI_ICON1"   -1, 8, 8, 0, 0
<79>     CTEXT            "Win32 Demo Program" -1, 0, 36, 160, 8
<80>     CTEXT            "(c) Игорь Виктор. 1999" -1, 0, 48, 160, 8
<81>     DEFPUSHBUTTON    "OK" IDOK, 64, 60, 32, 14, WS_GROUP
<82> }

```

В файле определены ресурсы трех типов: значок (ICON), меню (MENU) и диалог (DIALOG). Они могли быть созданы по отдельности редактором ресурсов, а затем с помощью текстового редактора объединены в один файл. При этом вы не должны забывать и про сопутствующие им включаемые файлы с расширением .h. Их также удобно объединить в единый файл так, как это сделано для нашего примера (листинг 18.10).

Листинг 18.10. Включаемый файл с идентификаторами элементов ресурсов prg19_4.h

```

#define IDM_DRAWTEXT      100
#define IDM_TEXTOUT       101
#define IDM_LENGTH        102
#define IDM_RECTANGLE     103
#define IDM_PEACOCK        104
#define IDM_LACES         105
#define IDM_ABOUT         106

#define IDC_EDIT1          1000

```

```
#define IDC_EDIT2    1001
#define IDC_EDIT3    1002
#define IDC_EDIT4    1003
#define IDC_STATIC   -1
```

На основании включаемого файла должен быть составлен эквивалентный включаемый файл `prg19_4.inc` (листинг 18.11). Директиву `include prg19_4.inc` необходимо помещать в начале исходного текста приложения. В принципе, этого можно и не делать, но тогда нельзя будет использовать символические имена констант, определенные в файле `prg19_4.h`. Вместо них в соответствующих местах программы придется использовать их численные значения.

Листинг 18.11. Включаемый файл с идентификаторами элементов ресурсов prg19_4.inc

```
IDM_DRAWTEXT    equ 100
IDM_TEXTOUT     equ 101
IDM_LENGTH      equ 102
IDM_RECTANGLE   equ 103
IDM_PEACOCK     equ 104
IDM_LACES        equ 105
IDM_ABOUT        equ 106
IDC_EDIT1        equ 1000
IDC_EDIT2        equ 1001
IDC_EDIT3        equ 1002
IDC_EDIT4        equ 1003
IDC_STATIC       equ -1
```

После того как файлы созданы, необходимо выполнить компиляцию файла ресурсов и получить его двоичный эквивалент `prg19_4.res`. Файл ресурсов рассматриваемого нами приложения, в отличие от файла ресурсов `prg19_3.rc`, имеет особенности. Эти особенности связаны с тем, что при описании ресурса окна диалога используются символические имена констант, определенные в файле `windows.h`. Дальнейшие наши действия будут зависеть от того, каким компилятором с языка C/C++ вы располагаете. Последовательность этих действий будет приблизительно одинаковой, отличным будет используемое программное обеспечение. Покажем их на примере Visual C++ версии 4.2. Программные средства именно этого пакета были использованы для компиляции файла ресурсов приложения `prg19_4`. Вариантов ваших действий может быть несколько, наиболее привлекательным является следующий¹:

- скопировать исполняемый файл компилятора ресурсов `..\Msdev\bin\rc.exe` в свой рабочий директорий `..\..\WORK32`;
- поместить в файл `autoexec.bat` строку: `call TasmVars.bat`. Файл `TasmVars.bat` предназначен для установки переменных окружения. Значения переменных определяют путь для поиска включаемых, исполняемых и других файлов. После модификации файла `autoexec.bat` необходимо перезагрузить компьютер, с тем, чтобы изменения вступили в силу.

¹ Предполагается, что вы следуете рекомендациям и ведете всю текущую работу в рабочем каталоге, к примеру `..\WORK`. В нем находятся необходимые файлы из пакета TASM и все файлы, относящиеся к текущему разрабатываемому приложению. Теперь у вас будет два каталога: для 16-разрядных приложений `..\WORK` и для 32-разрядных приложений `..\WORK32`.

○ запустить на выполнение компилятор ресурсов, указав ему в качестве параметра имя созданного нами ранее ресурсного файла `rc.exe prg19_4.rc`.

Если указанные выше действия были выполнены корректно, то в результате вы получите файл `prg19_4.res`. То, насколько удачно вам удалось определить ресурсы для вашего приложения, можно проверить только на этапе его выполнения. Если что-то вас не устраивает, то описанный выше процесс придется повторить, внеся необходимые корректировки.

Диалоговая процедура

В процессе работы с диалоговым окном пользователь выполняет некоторые действия, о которых с помощью механизма сообщений становится известно приложению. В приложении для каждого диалогового окна должна существовать своя процедура, предназначенная для обработки сообщений этого окна. Эта процедура называется *диалоговой процедурой*. Даже самое примитивное окно диалога содержит элемент, сообщение от которого поступает в диалоговую процедуру. Обычно такое окно содержит кнопку «OK» или «Cancel». На языке C/C++ соответствующая диалоговая процедура имеет вид, показанный в листинге 18.12.

Листинг 18.12. Диалоговая процедура на языке C/C++

```
BOOL CALLBACK DialogProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
        case WM_INITDIALOG:
            return 1;
        case WM_COMMAND:
            switch (LOWORD (wParam))
            {
                case IDOK: //или IDCANCEL
                    EndDialog(hwnd, 0);
                    return 1;
            }
    }
    return 0;
}
```

В приложении, текст которого показан в листинге 18.13, имеется фрагмент (строки 523–549), являющийся эквивалентом программы листинга 18.12 на языке ассемблера. Это процедура `AboutDialog`, поддерживающая работу диалогового окна `AboutBox`, описанного в файле ресурсов (см. листинг 18.9). Заметьте, что если оконная процедура самостоятельно обрабатывает сообщение, то она должна возвратить 1 (`return 1`), если нет, то 0 (`return 0`).

Окна диалога, имеющие большое количество элементов управления, должны, соответственно, иметь диалоговые процедуры, обрабатывающие сообщения от этих элементов.

В нашем примере определены два окна диалога, внешне очень похожие друг на друга. С помощью этих окон пользователь может задавать координаты начала и конца отрезка и углов прямоугольника. На рис. 18.3 приведен вид окна приложения с окном диалога для задания координат прямоугольника.

При задании координат пользователь сам должен контролировать правильность ввода данных, так как алгоритм программы не предусматривает проверки данных, вводимых в элементы ввода диалогового окна. Правильные значения, вводимые в каждое из полей ввода, должны содержать все четыре десятичные цифры, с включением, при необходимости, ведущих нулей.

Для обработки сообщений от этих диалоговых окон программа prg18.4.asm (см. листинг 18.13) содержит две диалоговые процедуры DialogProc1 и DialogProc2. Процедура DialogProc1 (строки 357–439) предназначена для ввода координат отрезка. Процедура DialogProc2 (строки 440–522) предназначена для ввода координат прямоугольника. Структура диалоговой процедуры аналогична структуре оконной процедуры. В начале диалоговой процедуры находится код, определяющий тип поступившего сообщения и, в зависимости от него, передающий управление в определенную точку диалоговой процедуры:

```
mov    eax, @message
cmp    ax, WM_INITDIALOG
je     @@wminitdialog
cmp    ax, WM_COMMAND
jne   @@exit_false
mov    ebx, @@wparam      ;в bx идентификатор элемента управления
cmp    bx, IDOK
je     @@idok
cmp    bx, IDCANCEL
je     @@idcancel
jmp   @@exit_false
```

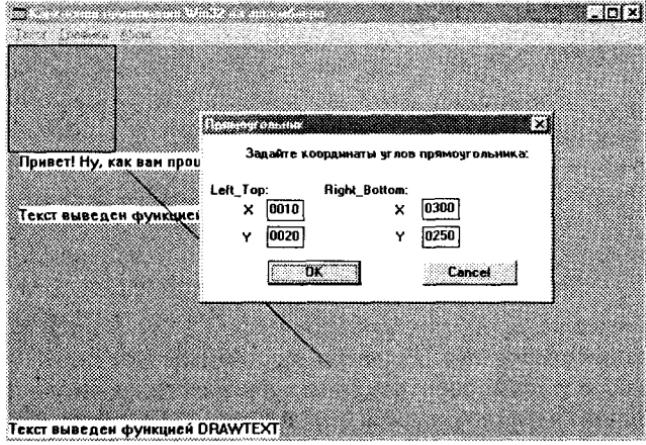


Рис. 18.3. Окно диалога для задания координат прямоугольника

Главное сообщение, поступающее в оконную процедуру, — сообщение WM_COMMAND. Именно оно несет информацию о действиях пользователя над элементами управления окна диалога. Так, в строках 361–371 листинга 18.13, обрабатывается два сообщения: WM_INITDIALOG и WM_COMMAND. Сообщение WM_INITDIALOG приходит в диалоговую процедуру один раз. Это происходит в процессе инициализации диалога перед появлением окна диалога в области окна приложения. Далее в диалоговую процедуру поступает последовательность сообщений WM_COMMAND. Параметр wParam

этого сообщения содержит идентификатор того элемента управления, над которым пользователь произвел некоторое действие, например, щелкнул кнопку «OK» или «Cancel» (рис. 18.3). Обратите внимание, что при описании элементов различных окон диалога используются одни и те же идентификаторы. Более того, в файле `prg19_4.h` этим идентификаторам назначены одни и те же константы. Именно их значения передаются в структуре сообщения для идентификации конкретного элемента окна диалога. Одинаковые значения не вносят никакой путаницы в работу приложения, так как для каждого окна диалога существует своя процедура.

Читателю полезно разобраться в деталях реализации диалоговых процедур `DialogProc1` и `DialogProc2`. Они используют две отладочные макрокоманды `sim4_to_EAXbin` и `show_eax`.

Макрокоманда `sim4_to_EAXbin` описана в программе `prg19_4.asm` и предназначена для преобразования строки из четырех символов десятичных цифр в эквивалентное двоичное число, помещаемое в регистр `eax`. В данной программе эта макрокоманда используется для преобразования символьных строк, считываемых из окон ввода диалога. Эти символьные строки логически представляют собой координаты отрезка и прямоугольника. Преобразованные с помощью макрокоманды `sim4_to_EAXbin` значения этих координат используются для работы функций Win32 API `MoveToEx` (строки 272–277), `LineTo` (строки 278–282) и `Rectangle` (строки 297–316). В конце своей работы макрокоманда `sim4_to_EAXbin` с помощью окна, выводимого функцией `MessageBox`, сообщает о результатах преобразования. Это отладочный момент работы и, учитывая учебный характер разрабатываемого приложения, он не убран из конечной его версии.

Макрокоманда `show_eax` также используется для отладки и является аналогом макрокоманды `show`, описанной в уроке 13. Попытка запустить `show` в нашем приложении ни к чему хорошему не приведет, поэтому для динамического контроля работы программы макрокоманда `show_eax` может оказаться очень полезной. Кстати, проблема отладки Windows-приложения достаточно актуальна, и для ее решения подчас приходится применять самые нетрадиционные решения, но это уже тема для отдельного разговора.

Активизация диалога

Для того чтобы отобразить диалоговое окно на экране и выполнить с его помощью некоторую работу, используется функция `DialogBoxParamA`. Эта функция имеет следующий формат (в нотации C/C++):

```
int DialogBoxParam (HINSTANCE hInstance, //дескриптор приложения
    LPCTSTR lpTemplateName, //указатель на строку с заголовком окна
    HWND hWndParent, //дескриптор окна
    DLGPROC lpDialogFunc, //указатель на диалоговую процедуру
    LPARAM dwInitParam) //значение, передаваемое в диалоговую функцию через lParam
```

В программе `prg19_4.asm` (см. листинг 18.13) обращение к этой функции производится трижды (строки 271, 296 и 344 в теле процедуры `MenuProc`) при выборе соответствующих пунктов меню. С помощью этой функции в диалоговую процедуру передается 32-битное значение, которое извлекается посредством параметра `lParam`.

По окончании работы окно диалога должно быть закрыто функцией `EndDialog`, имеющей формат:

```
BOOL EndDialog (HWND hDlg, //дескриптор окна диалога
    int nResult); //возвращаемое значение
```

Обычно закрытие окна производится как реакция на нажатие кнопок «OK» и (или) «Cancel».

Ниже приведен листинг 18.13, содержащий фрагменты текста приложения `prg18_4.asm`. Полностью оно содержится на гибком диске в каталоге данного урока.

Листинг 18.13. Фрагменты приложения `prg19_4.asm`

```
<1> :prg19_4.asm
<2> ;Пример приложения для Win32 с использованием меню,
<3> ;окон диалогов, решением проблемы перерисовки содержимого окна
<4> ;и демонстрацией некоторых принципов работы с графикой
<5> .486
<6> locals ;разрешает применение локальных меток (с префиксом @@) в программе
<7> .model flat, STDCALL ;модель памяти flat,
<8> ;STDCALL – передача параметров в стиле С (справа налево),
<9> ;вызываемая процедура чистит за собой стек
<10> include windowA.inc ;включаемый файл с описаниями базовых структур
                                и констант Win32
<11> include     prg19_4.inc      ;включаемый файл с определением имен
                                идентификатор меню и диалога
<12> ;Объявление внешними используемых в данной программе функций Win32 (ASCII):
<13> extrn      GetModuleHandleA:PROC
<14> extrn      GetVersionExA:PROC
<15> extrn      GetCommandLineA:PROC
<16> extrn      GetEnvironmentStringsA:PROC
<17> extrn      GetEnvironmentStringsA:PROC
<18> extrn      GetStartupInfoA:PROC
<19> extrn      LoadIconA:PROC
<20> extrn      LoadCursorA:PROC
<21> extrn      GetStockObject:PROC
<22> extrn      RegisterClassExA:PROC
<23> extrn      CreateWindowExA:PROC
<24> extrn      ShowWindow:PROC
<25> extrn      UpdateWindow:PROC
<26> extrn      GetMessageA:PROC
<27> extrn      TranslateMessage:PROC
<28> extrn      DispatchMessageA:PROC
<29> extrn      ExitProcess:PROC
<30> extrn      PostQuitMessage:PROC
<31> extrn      DefWindowProcA:PROC
<32> extrn      PlaySoundA:PROC
<33> extrn      ReleaseDC:PROC
<34> extrn      TextOutA:PROC
<35> extrn      GetDC:PROC
<36> extrn      BeginPaint:PROC
<37> extrn      EndPaint:PROC
<38> extrn      MessageBoxA:PROC
<39> extrn      DrawTextA:PROC
```

```

<40> extrn    GetClientRect:PROC
<41> extrn    GetSystemMetrics:PROC
<42> extrn    CreateCompatibleDC:PROC
<43> extrn    CreateCompatibleBitmap:PROC
<44> extrn    SelectObject:PROC
<45> extrn    GetStockObject:PROC
<46> extrn    PatBlt:PROC
<47> extrn    BitBlt:PROC
<48> extrn    InvalidateRect:PROC
<49> extrn    DeleteDC:PROC
<50> extrn    LineTo:PROC
<51> extrn    Rectangle:PROC
<52> extrn    MoveToEx:PROC
<53> extrn    DialogBoxParamA:PROC
<54> extrn    EndDialog:PROC
<55> extrn    GetDlgItemTextA:PROC
<56> :объявление оконной и диалоговых функций объектами, видимыми за пределами
           данного кода

<57> public     WindowProc
<58> public     DialogProc1
<59> public     DialogProc2
<60> .data
<61> Xstart      dd      0, 0
<62> Xend        dd      0, 0
<63> Ystart      dd      0, 0
<64> Yend        dd      0, 0
<65> hwnd         dd      0
<66> hInst        dd      0
<67> memdc        dd      0      ;!!!это глобальная переменная
<68> maxX          dd      0      ;!!!это глобальная переменная
<69> maxY          dd      0      ;!!!это глобальная переменная
<70> :lpVersionInformation OSVERSIONINFO      <?>
<71> wcl          WNDCLASSEX    <?>
<72> message      MSG      <?>
<73> ps           PAINTSTRUCT   <?>
<74> lpRect        RECT     <?>
<75> pt           POINT    <?>
<76> szClassName db      'Приложение Win32', 0
<77> szTitleName db      'Каркасное приложение Win32 на ассемблере', 0
<78> MesWindow    db      'Привет! Ну как вам процесс разработки приложения
                           на ассемблере?' 
<79> MesWindowLen =      $-MesWindow
<80> ;звуковые файлы
<81> playFileCreate db      'create.wav', 0
<82> playFilePaint db      'paint.wav', 0
<83> playFileDestroy db      'destroy.wav', 0
<84> ;имена ресурсов:
<85> lpmenu       db      "MYMENU", 0
<86> lpdlg1        db      "IDD_DIALOG1", 0
<87> lpdlg2        db      "IDD_DIALOG2", 0
<88> lpdlg3        db      "AboutBox", 0

```

```

<89> ;переменные для макроса show_eax
<90> eedx      dd      0
<91> eecx      dd      0, 0
<92> template   db      '0123456789ABCDEF'
<93> MesMsgBox  db      'Отладка (содержимое eax): ', 0
<94> ;описание макрокоманд
<95> include    show_eax.inc
<96> sim4_to_EAXbin  macro sim4:req
<97>           local  m1
<98>           push   eax
<99>           push   ebx
<100>          push   ecx
<101>          mov    ebx, 1
<102>          mov    eax, sim4
<103>          bswap  eax
<104>          mov    sim4, 0
<105>          push   eax
<106>          mov    ecx, 4
<107> m1:       and   eax, 0fh
<108>          imul  eax, ebx
<109>          imul  ebx, 10
<110>          add   sim4, eax
<111>          pop   eax
<112>          shr   eax, 8
<113>          push  eax
<114>          loop  m1
<115>          pop   eax
<116>          pop   ecx
<117>          pop   ebx
<118>          pop   eax
<119>          endm
<120> .code
<121> start     proc    near
<122> ;точка входа в программу:
<123> ;начало стартового кода
<124> .....
<125> ;конец стартового кода
<126> WinMain:
<127> ;определить класс окна ATOM RegisterClassEx(CONST WNDCLASSEX *lpWndClassEx),
<128> ;      где *lpWndClassEx – адрес структуры WndClassEx
<129> ;.....
<130>           call   RegisterClassExA
<131>           test  ax, ax ;проверить на успех регистрации класса окна
<132>           jz    end_cycl_msg ;неудача
<133> ;создаем окно:
<134> ;.....
<135>           call   CreateWindowExA
<136>           mov   hwnd, eax ;hwnd – дескриптор окна
<137> ;показать окно:
<138> ;.....
<139>           call   ShowWindow

```

```
<140> ;перерисовываем содержимое окна
<141> ;.....
<142>     call      UpdateWindow
<143> ;запускаем цикл сообщений:
<144> ;.....
<145> cycl_msg:
<146> ;.....
<147>     call      GetMessageA
<148>     cmp       ax, 0
<149>     je        end_cycl_msg
<150> ;трансляция ввода с клавиатуры
<151> ;.....
<152>     call      TranslateMessage
<153> ;отправим сообщение оконной процедуре
<154> ;.....
<155>     call      DispatchMessageA
<156>     jmp       cycl_msg
<157> end_cycl_msg:
<158> ;выход из приложения
<159> ;.....
<160>     call      ExitProcess
<161> start    endp
<162>
<163> ;-----WindowProc-----
<164> WindowProc proc
<165> arg      @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
<166> uses ebx, edi, esi, ebx      ;эти регистры обязательно должны сохраняться
<167> local    @@hdc:DWORD, @@hbrush:DWORD, @@hbit:DWORD
<168>     cmp       @@mes, WM_DESTROY
<169>     je        wmdestroy
<170>     cmp       @@mes, WM_CREATE
<171>     je        wmcREATE
<172>     cmp       @@mes, WM_PAINT
<173>     je        wmpaint
<174>     cmp       @@mes, WM_COMMAND
<175>     je        wmcCOMMAND
<176>     jmp       default
<177> wmcCREATE:
<178> ;создание растрового изображения, совместимого с окном приложения
<179> ;.....
<180> ;обозначим создание окна звуковым эффектом
<181> ;.....
<182>     call      PlaySoundA
<183> ;возвращаем значение 0
<184>     mov       eax, 0
<185>     jmp       exit_wndproc
<186> wmpaint:
<187> ;.....
<188> ;обозначим перерисовку окна звуковым эффектом
<189> ;.....
<190> ;вывод виртуального окна в реальное окно
<191> ;.....
```

Листинг 18.13 (продолжение)

```
<192> wmdestroy:  
<193> ;удалить виртуальное окно DeleteDC(memdc)  
<194> ;...  
<195> ;послать сообщение WM_QUIT  
<196> ;...  
<197> wmcancel:  
<198> ;вызов процедуры обработки сообщений от меню  
<199> ;MenuProc (DWORD @@hwnd, DWORD @@wparam)  
<200>         push    @@wparam  
<201>         push    @@hwnd  
<202>         call    MenuProc  
<203>         jmp     exit_wndproc  
<204> default:  
<205> ;обработка по умолчанию  
<206> ;...  
<207>         jmp exit_wndproc  
<208> ;...  
<209> exit_wndproc:  
<210>         ret  
<211> WindowProc endp  
<212> ;-----MenuProc-----  
<213> ;обработка сообщений от меню  
<214> MenuProc proc  
<215> arg      @@hwnd:DWORD, @@wparam:DWORD  
<216> uses    eax, ebx  
<217>         mov     ebx, @@wparam      ;в bx идентификатор меню  
<218>         cmp     bx, IDM_DRAWTEXT  
<219>         je      @@idmdrawtext  
<220>         cmp     bx, IDM_TEXTOUT  
<221>         je      @@idmtextout  
<222>         cmp     bx, IDM_LENGTH  
<223>         je      @@idmlength  
<224>         cmp     bx, IDM_RECTANGLE  
<225>         je      @@idmrectangle  
<226>         cmp     bx, IDM_PEACOCK  
<227>         je      @@idmpeacock  
<228>         cmp     bx, IDM_LACES  
<229>         je      @@idmlaces  
<230>         cmp     bx, IDM_ABOUT  
<231>         je      @@idmabout  
<232>         jmp     @@exit  
<233> @@idmdrawtext:  
<234> ;получим размер рабочей области BOOL GetClientRect(HWND hWnd, LPRECT lpRect);  
<235>         push    offset lpRect  
<236>         push    @@hwnd  
<237>         call    GetClientRect  
<238> ;выведем строку текста в окно int DrawText(HDC hDC, LPCTSTR lpString, int nCount,  
<239> ; LPRECT lpRect, UINT uFormat);  
<240>         push    DT_SINGLELINE+DT_BOTTOM  
<241>         push    offset lpRect
```

```
<242>         push    -1
<243>         push    offset @@TXT_DRAWTEXT
<244>         push    memdc
<245>         call    DrawTextA
<246> ;генерация сообщения WM_PAINT для вывода строки на экран
<247>         push    1
<248>         push    NULL
<249>         push    @@hwnd
<250>         call    InvalidateRect
<251>         jmp    @@exit
<252> @@idmtextout:
<253> ;выведем строку текста в окно BOOL TextOut(HDC hdc, int nXStart,
<254> ;                           int nYStart, LPCTSTR lpString, int cbString);
<255> ;.....
<256>         call    TextOutA
<257> ;генерация сообщения WM_PAINT для вывода строки на экран
<258>         push    0
<259>         push    NULL
<260>         push    @@hwnd
<261>         call    InvalidateRect
<262>         jmp    @@exit
<263> @@idmlength:
<264> ;вызываем диалоговое окно int DialogBoxParam(HINSTANCE hInstance, LPCTSTR
<265> ;                           1pTemplateName,
<266>         push    HWND hWndParent, DLGPROC 1pDialogFunc, LPARAM dwInitParam)
<267>         push    0
<268>         push    offset DialogProc1
<269>         push    @@hwnd
<270>         push    offset 1pdlg1
<271>         push    hInst
<272>         call    DialogBoxParamA
<273> ;установить текущую точку BOOL MoveToEx(HDC hdc, int X, int Y, LPOINT
<274> ;                           1pPoint)
<275>         push    NULL
<276>         push    Ystart
<277>         push    Xstart
<278>         push    memdc
<279>         call    MoveToEx
<280> ;вывод линии BOOL LineTo(HDC hdc, int nXEnd, int nYEnd)
<281>         push    Yend
<282>         push    Xend
<283>         push    memdc
<284>         call    LineTo
<285> ;генерация сообщения WM_PAINT для вывода строки на экран
<286>         push    0
<287>         push    NULL
<288>         push    @@hwnd
<289>         call    InvalidateRect
<290>         jmp    @@exit
<291> @@idmrectangle:
<292> ;вызываем диалоговое окно
<293>         push    0
```

Листинг 18.13 (продолжение)

```
<292>     push    offset DialogProc2
<293>     push    @@hwnd
<294>     push    offset 1pd1g2
<295>     push    hInst
<296>     call    DialogBoxParamA
<297> ; вывод прямоугольника BOOL Rectangle(HDC hdc, int nLeftRect,
<298> :           int nTopRect, int nRightRect, int nBottomRect)
<299>         push    Ystart
<300>         pop     eax
<301>         push    eax
<302>         show_eax
<303>         push    Xstart
<304>         pop     eax
<305>         push    eax
<306>         show_eax
<307>         push    Yend
<308>         pop     eax
<309>         push    eax
<310>         show_eax
<311>         push    Xend
<312>         pop     eax
<313>         push    eax
<314>         show_eax
<315>         push    memdc
<316>         call    Rectangle
<317> ; генерация сообщения WM_PAINT для вывода строки на экран
<318>         push    0
<319>         push    NULL
<320>         push    @@hwnd
<321>         call    InvalidateRect
<322>         jmp    @@exit
<323> @@idmpeacock:
<324>         push    MB_ICONINFORMATION+MB_OK
<325>         push    offset szTitleName
<326>         push    offset @@TXT_PEACOCK
<327>         push    @@hwnd
<328>         call    MessageBoxA
<329>         jmp    @@exit
<330> @@idmlaces:
<331>         push    MB_ICONINFORMATION+MB_OK
<332>         push    offset szTitleName
<333>         push    offset @@TXT_LACES
<334>         push    @@hwnd
<335>         call    MessageBoxA
<336>         jmp    @@exit
<337> @@idmabout:
<338> ; вызываем диалоговое окно
<339>         push    0
<340>         push    offset AboutDialog
<341>         push    @@hwnd
<342>         push    offset 1pd1g3
```

```

<343>         push    hInst
<344>         call    DialogBoxParamA
<345>         jmp    @@exit
<346> ;.....
<347> @@exit:
<348>             mov    eax, 0
<349>             ret
<350> @eTXT_ABOUT      db     'IDM_ABOUT', 0
<351> @eTXT_LACES       db     'IDM_LACES', 0
<352> @eTXT_PEACOCK     db     'IDM_PEACOCK', 0
<353> @eTXT_TEXTOUT     db     'Текст выведен функцией TEXTOUT'
<354> lenTXT_TEXTOUT=$-@eTXT_TEXTOUT
<355> @eTXT_DRAWTEXT    db     'Текст выведен функцией DRAWTEXT', 0
<356> MenuProc        endp
<357> ;-----DialogProc1-----
<358> DialogProc1 proc
<359> arg      @hd1g:DWORD, @message:DWORD, @wparam:DWORD, @lparam:DWORD
<360> uses     eax, ebx, edi, esi
<361>         mov    eax, @message
<362>         cmp    ax, WM_INITDIALOG
<363>         je     @@wminitdialog
<364>         cmp    ax, WM_COMMAND
<365>         jne    @@exit_false
<366>         mov    ebx, @wparam : в bx идентификатор элемента управления
<367>         cmp    bx, IDOK
<368>         je     @@idok
<369>         cmp    bx, IDCANCEL
<370>         je     @@idcancel
<371>         jmp    @@exit_false
<372> @@wminitdialog:
<373>         jmp    @@exit_true
<374> @@idok:
<375> ;прочитаем Xstart UINT GetDlgItemText(HWND hDlg, int nIDDlgItem,
<376> ;                           LPTSTR lpString, int nMaxCount);
<377>         push    5
<378>         push    offset Xstart
<379>         push    IDC_EDIT1
<380>         push    @hd1g
<381>         call    GetDlgItemTextA
<382>         push    MB_ICONINFORMATION+MB_OK
<383>         push    offset szTitleName
<384>         push    offset Xstart
<385>         push    @hd1g
<386>         call    MessageBoxA
<387>         sim4_to_EAXbin Xstart
<388> ;прочитаем Ystart
<389>         push    5
<390>         push    offset Ystart
<391>         push    IDC_EDIT2
<392>         push    @hd1g
<393>         call    GetDlgItemTextA
<394>         push    MB_ICONINFORMATION+MB_OK

```

```

<395>      push    offset szTitleName
<396>      push    offset Ystart
<397>      push    @@hdlg
<398>      call    MessageBoxA
<399>      sim4_to_EAXbin Ystart
<400> ;прочитаем Xend
<401>      push    5
<402>      push    offset Xend
<403>      push    IDC_EDIT3
<404>      push    @@hdlg
<405>      call    GetDlgItemTextA
<406>      push    MB_ICONINFORMATION+MB_OK
<407>      push    offset szTitleName
<408>      push    offset Xend
<409>      push    @@hdlg
<410>      call    MessageBoxA
<411>      sim4_to_EAXbin Xend
<412> ;прочитаем Yend
<413>      push    5
<414>      push    offset Yend
<415>      push    IDC_EDIT4
<416>      push    @@hdlg
<417>      call    GetDlgItemTextA
<418>      push    MB_ICONINFORMATION+MB_OK
<419>      push    offset szTitleName
<420>      push    offset Yend
<421>      push    @@hdlg
<422>      call    MessageBoxA
<423>      sim4_to_EAXbin Yend
<424>      push    0
<425>      push    @@hdlg
<426>      call    EndDialog
<427>      jmp    @@exit_true
<428> @@idcancel:
<429>      push    NULL
<430>      push    @@hdlg
<431>      call    EndDialog
<432>      jmp    @@exit_true
<433> @@exit_false:
<434>      mov     eax, 0
<435>      ret
<436> @@exit_true:
<437>      mov     eax, 1
<438>      ret
<439> DialogProc1 endp
<440> ;-----DialogProc2-----
<441> DialogProc2 proc
<442> arg    @@hdlg:DWORD, @message:DWORD, @wparam:DWORD, @lparam:DWORD
<443> uses   eax, ebx, edi, esi
<444> mov     eax, @message
<445> cmp     ax, WM_INITDIALOG
<446> je     @@wminitdialog

```

```
<447>     cmp    ax, WM_COMMAND
<448>     jne    @@exit_false
<449>     mov    ebx, @@wparam ; в bx идентификатор элемента управления
<450>     cmp    bx, IDOK
<451>     je     @@idok
<452>     cmp    bx, IDCANCEL
<453>     je     @@idcancel
<454>     jmp    @@exit_false
<455> :@wmInitDialog:
<456>     jmp    @@exit_true
<457> @@idok:
<458> :прочитаем Xstart UINT GetDlgItemText(HWND hDlg, int nIDDlgItem,
<459> :           LPTSTR lpString, int nMaxCount);
<460>     push   5
<461>     push   offset Xstart
<462>     push   IDC_EDIT1
<463>     push   @@hd1g
<464>     call   GetDlgItemTextA
<465>     push   MB_ICONINFORMATION+MB_OK
<466>     push   offset szTitleName
<467>     push   offset Xstart
<468>     push   @@hd1g
<469>     call   MessageBoxA
<470>     sim4_to_EAXbin Xstart
<471> :прочитаем Ystart
<472>     push   5
<473>     push   offset Ystart
<474>     push   IDC_EDIT2
<475>     push   @@hd1g
<476>     call   GetDlgItemTextA
<477>     push   MB_ICONINFORMATION+MB_OK
<478>     push   offset szTitleName
<479>     push   offset Ystart
<480>     push   @@hd1g
<481>     call   MessageBoxA
<482>     sim4_to_EAXbin Ystart
<483> :прочитаем Xend
<484>     push   5
<485>     push   offset Xend
<486>     push   IDC_EDIT3
<487>     push   @@hd1g
<488>     call   GetDlgItemTextA
<489>     push   MB_ICONINFORMATION+MB_OK
<490>     push   offset szTitleName
<491>     push   offset Xend
<492>     push   @@hd1g
<493>     call   MessageBoxA
<494>     sim4_to_EAXbin Xend
<495> :прочитаем Yend
<496>     push   5
<497>     push   offset Yend
<498>     push   IDC_EDIT4
<499>     push   @@hd1g
```

Листинг 18.13 (продолжение)

```
<500>      call   GetDlgItemTextA
<501>      push   MB_ICONINFORMATION+MB_OK
<502>      push   offset szTitleName
<503>      push   offset Yend
<504>      push   @@hdlg
<505>      call   MessageBoxA
<506>      sim4_to_EAXbin Yend
<507>      push   NULL
<508>      push   @@hdlg
<509>      call   EndDialog
<510>      jmp   @@exit_true
<511> @@idcancel:
<512>      push   NULL
<513>      push   @@hdlg
<514>      call   EndDialog
<515>      jmp   @@exit_true
<516> @@exit_false:
<517>      mov    eax, 0
<518>      ret
<519> @@exit_true:
<520>      mov    eax, 1
<521>      ret
<522> DialogProc2 endp
<523> ;-----AboutDialog-----
<524> AboutDialog proc
<525> arg   @@hdlg:DWORD, @@message:DWORD, @@wparam:DWORD, @@lparam:DWORD
<526> uses  eax, ebx, edi, esi
<527>      mov    eax, @@message
<528>      cmp    ax, WM_INITDIALOG
<529>      je    @@wminitdialog
<530>      cmp    ax, WM_COMMAND
<531>      jne   @@exit_false
<532>      mov    ebx, @@wparam ; в bx идентификатор элемента управления
<533>      cmp    bx, IDOK
<534>      je    @@idok
<535>      jmp   @@exit_false
<536> @@wminitdialog:
<537>      jmp   @@exit_true
<538> @@idok:
<539>      push   NULL
<540>      push   @@hdlg
<541>      call   EndDialog
<542>      jmp   @@exit_true
<543> @@exit_false:
<544>      mov    eax, 0
<545>      ret
<546> @@exit_true:
<547>      mov    eax, 1
<548>      ret
<549> AboutDialog endp
<550> end   start
```

Возможно, вас испугал большой размер исходного текста приложения `prg19_4.asm`. Но он может быть существенно сокращен за счет использования макрокоманд и процедур. Если вы с их помощью реорганизуете исходный текст, то он станет таким же читабельным и удобным, как аналогичный текст на языке C/C++. Зато, в отличие от текста на C/C++, текст на ассемблере – готовая машинная программа, в которую ничего лишнего не добавляется и код которой полностью контролируется. Более того, над этим текстом можно проводить весь комплекс мероприятий по его оптимизации. Мы специально не стали делать ничего в плане улучшения внешнего вида исходного текста, с тем, чтобы не навязывать читателю свои подходы к этому процессу, тем более, что это могло его ввести в определенные заблуждения. Текст программы в настоящем виде более всего отражает суть процессов, происходящих в системе Windows во время исполнения приложения, и то, каким образом приложение взаимодействует с системой.

Нами остались неразработаны реакции приложения на выбор пользователем пунктов меню Графика ▶ Эффекты ▶ Павлин и Графика ▶ Эффекты ▶ Кружева. Их реализация требует использования команд сопроцессора и поэтому будет рассмотрена на следующем уроке.

Подведем некоторые итоги:

- Разработка Windows-приложения на языке ассемблера вполне реальное и в ряде случаев оправданное дело. Средства языка ассемблера поддерживают все необходимые для создания полноценного Windows-приложения действия.
- В процессе разработки Windows-приложения на языке ассемблера необходимо опираться на информацию и некоторые программные средства, предоставляемые одним из пакетов языка высокого уровня. Лучше всего для этой цели подходит пакет VC++ версии 4.0 и выше. Основную ценность предоставляют включаемые файлы, редактор ресурсов, работающий в составе интегрированной среды разработки, и компилятор ресурсов. Интерес могут представлять различные утилиты, входящие в состав пакета Visual C++, например Spy++.
- Приступить к разработке приложения для Windows на ассемблере лучше всего, имея некоторый опыт разработки приложений на языке высокого уровня. Это необходимо для понимания логики работы приложения. Когда понимание логики работы Windows-приложения достигнуто, выбор языка для его реализации приобретает в большей степени техническое значение и определяется постановкой задачи и предполагаемыми условиями ее эксплуатации.
- Овладев информацией этого урока, читатель при необходимости может легко разработать программу другого типа, поддерживаемую системой Windows, – консольное приложение (см. урок 20).

19

УРОК

Архитектура и программирование сопроцессора

-
- Архитектура
 - Форматы данных
 - Система команд
 - Исключения и их обработка
 - Использование отладчика
-

Важной частью архитектуры микропроцессоров Intel является наличие устройства для обработки числовых данных в формате с плавающей точкой. До этого момента мы рассматривали команды и алгоритмы обработки целочисленных данных (чисел с фиксированной точкой). Целочисленные форматы подробно обсуждались в материале уроков 2 и 8.

Устройства для обработки чисел с плавающей точкой появились в компьютерах давно. С точки зрения архитектуры они выглядели по-разному. Так, в архитектуре EC ЭВМ (аналог IBM 360/370) устройство с плавающей точкой было естественной частью этой архитектуры со своим регистровым пространством и подсистемой команд. Архитектура компьютеров на базе микропроцессоров вначале опиралась исключительно на целочисленную арифметику. С ростом мощи, а главное с осознанием разработчиками микропроцессорной техники того факта, что их устройства могут составить достойную конкуренцию своим «большим» предшественникам, в архитектуре компьютеров на базе микропроцессоров стали появляться устройства для обработки чисел с плавающей точкой. В архитектуре семейства микропроцессоров Intel 80x86 устройство для обработки чисел с плавающей точкой появилось в составе компьютера на базе микропроцессора i8086/88 и получило название *математический сопроцессор* (далее просто *сопроцессор*). Выбор такого названия был обусловлен тем, что, во-первых, это устройство было предназначено для расширения вычислительных возможностей основного процессора, а, во-вторых, оно было реализовано в виде отдельной микросхемы, то есть его присутствие было необязательным. Микросхема сопроцессора для микропроцессора i8086/88 имела название i8087. С появлением новых моделей микропроцессоров Intel совершенствовались и сопроцессоры, хотя их программная модель осталась практически неизменной. Как отдельные (а, соответственно, необязательные в конкретной комплектации компьютера) устройства, сопроцессоры сохранялись вплоть до модели микропроцессора i386 и имели название i287 и i387 соответственно. Начиная с модели i486, сопроцессор исполняется в одном корпусе с основным микропроцессором и, таким образом, является неотъемлемой частью компьютера.

Для чего нужен сопроцессор, какие возможности добавляет он к тому, что делает основной процессор, кроме возможности обработки еще одного формата данных? Перечислим некоторые из этих возможностей:

- полная поддержка стандартов IEEE-754 и 854 на арифметику с плавающей точкой. Эти стандарты описывают как форматы данных, с которыми должен работать сопроцессор, так и набор реализуемых им функций;
- поддержка численных алгоритмов для вычисления значений тригонометрических функций, логарифмов и т. п. Эта работа сопроцессора абсолютна прозрачна

для программиста, что само по себе очень ценно, так как не требует от него необходимости самостоятельной разработки соответствующих подпрограмм;

- обработка десятичных чисел с точностью до 18 разрядов, что позволяет сопроцессору выполнять арифметические операции без округления над целыми десятичными числами со значениями до 10^{18} (а мы с вами изошьрались на уроке 8, хотя, все же не совсем напрасно);

- обработка вещественных чисел из диапазона $3.37 \times 10^{-4932} \dots 1.18 \times 10^{+4932}$.

Нужно отметить, что в последние годы разработчики компьютерной периферии все активнее забирают часть вычислений у центрального процессора с целью более эффективной реализации специализированных операций. Очень ярко это проявляется на рынке мультимедийного оборудования, где множество разработчиков предлагают видеокарты с чипсетами (наборами микросхем), более эффективно реализующими работу с графикой, чем это делает сам микропроцессор. Нельзя не упомянуть здесь, что такая же ситуация сложилась и с широко рекламированной дополнительной группой команд микропроцессоров Pentium MMX (MultiMedia eXtensions) и Pentium II/III. Эту группу команд мы рассмотрим на следующем уроке. Пока она не оправдала надежд разработчиков мультимедиа-приложений. Несмотря на это, дополнение системы команд микропроцессора командами сопроцессора и MMX-расширения предоставляет ряд уникальных свойств и возможностей, пренебрегать которыми было бы опрометчиво.

Архитектура сопроцессора

На уроке 2 мы определили место сопроцессора в архитектуре компьютера. Аппаратная реализация сопроцессора нас интересует лишь в видимой для программиста части. Как и в случае с основным процессором, интерес для нас представляет *программная модель сопроцессора*. С точки зрения программиста, сопроцессор представляет собой совокупность регистров, каждый из которых имеет свое функциональное назначение (рис. 19.1).

В программной модели сопроцессора можно выделить три группы регистров.

1. Восемь регистров $r0 \dots r7$, составляющих основу программной модели сопроцессора — *стек сопроцессора*. Размерность каждого регистра 80 битов. Такая организация характерна для устройств, специализирующихся на обработке вычислительных алгоритмов. Вспомните, как представляются математические выражения с использованием обратной польской записи (ПОЛИЗ). Вычисление такого выражения заключается в выборке с вершины стека очередной операции. Если это двухместная операция, то с вершины стека снимаются два операнда, над которыми и производятся действия в соответствии со снятой ранее операцией. Более подробно представление выражения в форме ПОЛИЗ мы рассмотрим далее. Реализация численных алгоритмов на основе регистрового стека позволяет получить существенный выигрыш в скорости вычислений.

2. Три служебных регистра:

регистр состояния сопроцессора swr (Status Word Register — регистр слова состояния) — отражает информацию о текущем состоянии сопроцессора. В ре-

гистре *swr* содержатся поля, позволяющие определить: какой регистр является текущей вершиной стека сопроцессора, какие исключения возникли после выполнения последней команды, каковы особенности выполнения последней команды (некий аналог регистра флагов основного процессора) и т. д.;

управляющий регистр сопроцессора cwr (Control Word Register — регистр слова управления) — управляет режимами работы сопроцессора. С помощью полей в этом регистре можно регулировать точность выполнения численных вычислений, управлять округлением, маскировать исключения;

регистр слова тегов twr (Tags Word Register — слово тегов) — используется для контроля за состоянием каждого из регистров $r0 \dots r7$. Команды сопроцессора используют этот регистр, например, для того, чтобы определить возможность записи значений в эти регистры.

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

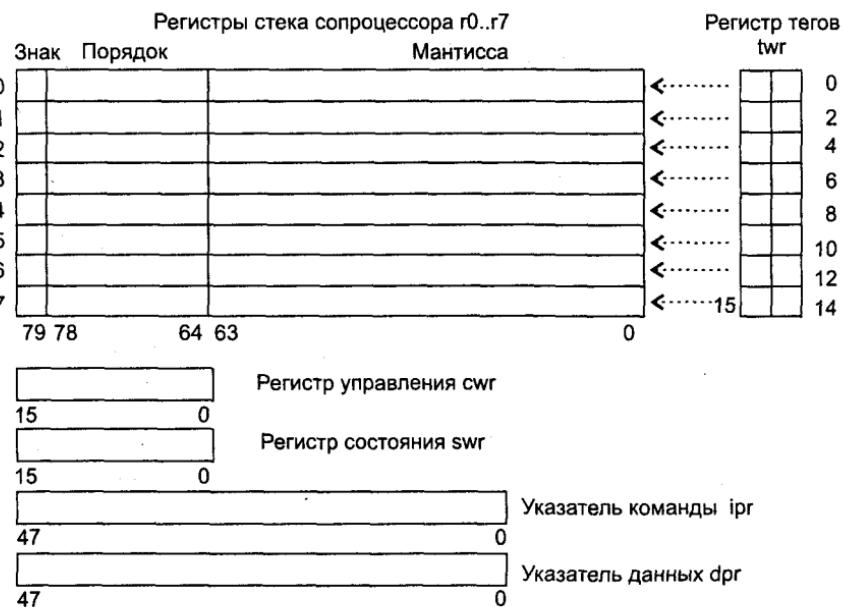


Рис. 19.1. Программная модель сопроцессора

Все эти регистры являются программно доступными. Однако к одним из них доступ получить достаточно легко, для этого в системе команд сопроцессора существуют специальные команды. К другим регистрам получить доступ сложнее, так как специальных команд для этого нет, поэтому необходимо выполнить дополнительные действия.

Рассмотрим общую логику работы сопроцессора и более подробно охарактеризуем перечисленные регистры.

Регистровый стек сопроцессора организован по принципу кольца. Это означает, что среди всех регистров, составляющих стек, нет такого, который является вершиной стека. Напротив, все регистры стека с функциональной точки зрения абсолютно одинаковы и равноправны. Но, как известно, в стеке всегда должна быть вершина. И она действительно есть, но является плавающей. Контроль текущей вершины осуществляется аппаратно с помощью трехбитового поля `top` регистра `swr` (рис. 19.2). В поле `top` фиксируется номер регистра стека 0...7 (`r0...r7`), являющегося в данный момент текущей вершиной стека.

Регистр состояния `swr`

| b | c3 | top | c2 | c1 | co | es | sf | pe | ue | oe | ze | de | ie | |
|----|----|-----|----|----|----|----|----|----|----|----|----|----|----|---|
| 15 | 14 | 13 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Рис. 19.2. Формат регистра состояния сопроцессора `swr`

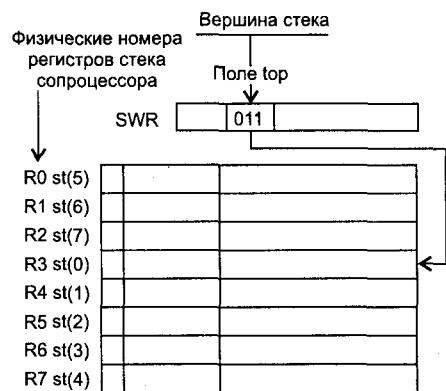
Команды сопроцессора не оперируют физическими номерами регистров стека `r0...r7`. Вместо этого они используют логические номера этих регистров `st(0)...st(1)`. С помощью логических номеров реализуется относительная адресация регистров стека сопроцессора. На рис. 19.3, б показан пример, когда текущей вершиной до записи в стек является физический регистр стека `r3`, а после записи в стек текущей вершиной становится физический регистр стека `r2`. То есть, по мере записи в стек, указатель его вершины движется по направлению к младшим номерам физических регистров (уменьшается на единицу). Если текущей вершиной является `r0`, то после записи очередного значения в стек сопроцессора его текущей вершиной станет физический регистр `r7` (рис. 19.3, а). Что касается логических номеров регистров стека `st(0)...st(1)`, то, как следует из рис. 19.3, они «плавают» вместе с изменением текущей вершины стека. Таким образом, реализуется принцип кольца.

На первый взгляд такая организация стека кажется странной. Но, как оказалось, она обладает большой гибкостью. Это хорошо видно на примере передачи параметров подпрограмме. Для повышения гибкости разработки и использования подпрограмм не желательно привязывать их по передаваемым параметрам к аппаратным ресурсам (физическим номерам регистров сопроцессора). Гораздо удобнее задавать порядок следования передаваемых параметров в виде логических номеров регистров. Такой способ передачи был бы однозначным и не требовал от разработчика знания лишних подробностей об аппаратных реализациях сопроцессора. Логическая нумерация регистров сопроцессора, поддерживаемая на уровне системы команд, идеально реализует эту идею. При этом не имеет значения, в какой физический регистр стека сопроцессора были помещены данные перед вызовом подпрограммы, определяющим является только порядок следования параметров в стеке. По этой причине подпрограмме важно знать уже не место, а только порядок размещения передаваемых параметров в стеке.

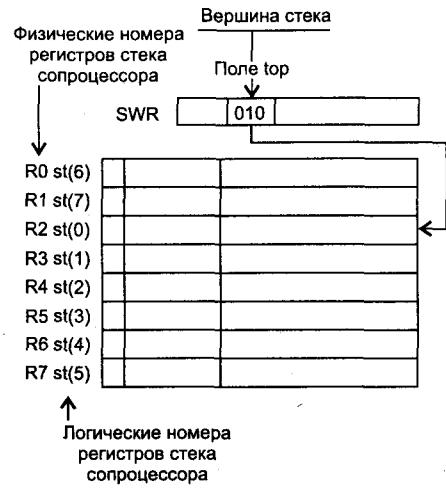
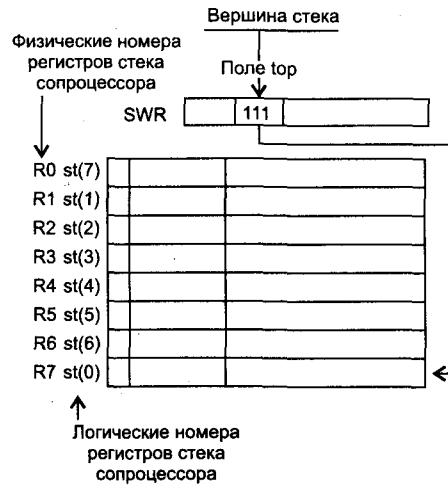
Перед тем как приступить к описанию команд и данных, с которыми работает сопроцессор, отметим, каким образом «уживаются» между собой эти два разных вычислительных устройства — процессор и сопроцессор. Каждое из этих устройств имеет свои, несовместимые друг с другом системы команд и форматы обрабатываемых данных. Несмотря на то что сопроцессор архитектурно представляет собой отдельное вычислительное устройство, он не может существовать от-

дельно от основного процессора. Выше уже было отмечено, что первые модели процессоров Intel (i8086, i286, i386) и сопроцессоров (соответственно i8087, i287, i387) выполнялись как отдельные устройства — сопроцессор, при необходимости, вставлялся в специальный разъем на системной плате. Начиная с модели i486, сопроцессор и основной процессор производятся в одном корпусе и являются физически неделимыми, то есть архитектурно это по-прежнему два разных устройства, а аппаратно — одно.

Состояние стека сопроцессора до выполнения операции записи в стек



Состояние стека сопроцессора после выполнения операции записи в стек



a

б

Рис. 19.3. Физическая и логическая нумерации регистров стека сопроцессора

Процессор и сопроцессор, являясь двумя самостоятельными вычислительными устройствами, могут работать параллельно. Но этот параллелизм касается толи

ко их внутренней работы над исполнением очередной команды. Как реализован этот параллелизм, в чем его суть и особенности? Оба процессора подключены к общей системной шине и имеют доступ к одинаковой информации. Инициирует процесс выборки очередной команды всегда основной процессор. После выборки команда попадает одновременно в оба процессора. Любая команда сопроцессора имеет код операции, первые пять бит, которого имеют значение 11011. Когда код операции начинается этими битами, то основной процессор по дальнейшему содержимому кода операции выясняет, требует ли данная команда обращения к памяти. Если это так, то основной процессор формирует физический адрес операнда и обращается к памяти, после чего содержимое ячейки памяти выставляется на шину данных. Если обращение к памяти не требуется, то основной процессор заканчивает работу над данной командой (не делая попытки ее исполнения!) и приступает к декодированию следующей команды из текущего входного командного потока. Что же касается сопроцессора, то выбранная команда, как уже было отмечено, попадает в него одновременно с основным процессором. Сопроцессор, определив по первым пяти битам, что очередная команда принадлежит его системе команд, начинает ее исполнение. Если команда требовала операнд в памяти, то сопроцессор обращается к шине данных за чтением содержимого ячейки памяти, которое к этому моменту предоставлено основным процессором. Из этой схемы взаимодействия следует, что в определенных случаях необходимо согласовывать работу обоих устройств. К примеру, если во входном потоке сразу за командой сопроцессора следует команда основного процессора, использующая результаты работы предыдущей команды, то сопроцессор не успеет выполнить свою команду за то время, когда основной процессор, пропустив сопроцессорную команду, выполнит свою. Очевидно, что логика работы программы будет нарушена. Возможна и другая ситуация. Если входной поток команд содержит последовательность из нескольких команд сопроцессора, то, очевидно, что процессор в отличие от сопроцессора проскочит их очень быстро, чего он не должен делать, так как он обеспечивает внешний интерфейс для сопроцессора. Эти и другие, более сложные, ситуации приводят к необходимости синхронизации между собой работы двух процессоров. В первых моделях микропроцессоров это делалось путем вставки перед или после каждой команды сопроцессора специальной команды `wait` или `fwait`. Работа данной команды заключалась в приостановке работы основного процессора до тех пор, пока сопроцессор не закончит работу над последней командой. В моделях микропроцессора (начиная с i486) подобная синхронизация выполняется командами `wait/fwait`, которые введены в алгоритм работы большинства команд сопроцессора. Но для некоторых команд из группы команд управления сопроцессором (см. далее) оставлена возможность выбора между командами с синхронизацией (ожиданием) и без нее.

Из всего сказанного можно сделать важный вывод: использование сопроцессора является совершенно прозрачным для программиста. В общем случае ему следует воспринимать сопроцессор как набор дополнительных регистров, для работы с которыми предназначены специальные команды. Для организации эффективной работы с сопроцессором программист должен хорошо разобраться со структурой регистров и логикой их использования. Поэтому перед тем как приступить к рассмотрению команд и данных, с которыми работает сопроцессор, приведем описание структуры некоторых регистров сопроцессора.

Регистр состояния swr

Выше было отмечено, что регистр `swr` отражает текущее состояние сопроцессора после выполнения последней команды. Структурно регистр `swr` (см. рис. 19.2) состоит из:

- 6 флагов исключительных ситуаций;
- бита `s`f (Stack Fault) — ошибка работы стека сопроцессора. Бит устанавливается в единицу, если возникает одна из трех исключительных ситуаций (см. ниже) — PE, UE или IE. В частности, его установка информирует о попытке записи в заполненный стек, или, напротив, попытке чтения из пустого стека. После того как вы проанализировали этот бит, его нужно снова установить в ноль, вместе с битами PE, UE или IE (если они были установлены);
- бита `e`s (Error Summary) — суммарная ошибка работы сопроцессора. Бит устанавливается в единицу, если возникает любая из шести перечисленных ниже исключительных ситуаций;
- четырех битов `c 0—c 3` (Condition Code) — кода условия. Назначение этих битов аналогично флагам в регистре `eflags` основного процессора — отразить результат выполнения последней команды сопроцессора. В Справочнике для некоторых команд сопроцессора приведена интерпретация битов `c 0—c 3`;
- трехбитного поля `top`. Поле содержит указатель регистра текущей вершины стека.

Почти половину регистра `swr` занимают биты (флаги) для регистрации исключительных ситуаций. На уроке 15 нами была введена классификация прерываний по месту их возникновения (внешние и внутренние). Внутренние прерывания возникают в ходе работы текущей программы и делятся на синхронные (по команде `int`) и асинхронные, называемые *исключениями* или *особыми случаями*. Таким образом, *исключения* — это разновидность прерываний, с помощью которых процессор информирует программу о некоторых особенностях ее реального исполнения. Сопроцессор также обладает способностью возбуждения подобных прерываний при возникновении определенных ситуаций (не обязательно ошибочных). Все возможные исключения сведены к шести типам, каждому из которых соответствует один бит в регистре `swr`. Программисту совсем не обязательно писать обработчик для реакции на ситуацию, приведшую к некоторому исключению. Сопроцессор умеет самостоятельно реагировать на многие из них. Это так называемая обработка исключений по умолчанию. Для того чтобы «заказать» сопроцессору обработку определенного типа исключения по умолчанию, необходимо это исключение замаскировать. Такое действие выполняется с помощью установки в единицу нужного бита в управляющем регистре сопроцессора `cwr` (рис. 19.4). Приведем типы исключений, фиксируемые с помощью регистра `swr`:

- `IE` (Invalide operation Error) — недействительная операция;
- `DE` (Denormalized operand Error) — денормализованный операнд;
- `ZE` (divide by Zero Error) — ошибка деления на нуль;
- `OE` (Overflow Error) — ошибка переполнения. Возникает в случае выхода порядка числа за максимально допустимый диапазон;

- UE (Underflow Error) — ошибка антипереполнения. Возникает, когда результат слишком мал;
- PE (Precision Error) — ошибка точности. Устанавливается, когда сопроцессору приходится округлять результат из-за того, что его точное представление невозможно. Так, сопроцессору (как и читателю) никогда не удастся точно разделить 10 на 3.

При возникновении любого из этих шести типов исключений устанавливается в единицу соответствующий бит в регистре `swr`, вне зависимости от того, было ли замаскировано это исключение в регистре `cwr` или нет. Более подробно об исключениях, в частности об условиях их возникновения, мы поговорим в конце занятия.

Регистр управления `cwr`

Регистр управления работой сопроцессора определяет особенности обработки численных данных (рис. 19.4). Он состоит из:

- шести масок исключений;
- поля управления точностью `pc` (Precision Control);
- поля управления округлением `rc` (Rounding Control).

Шесть масок предназначены для маскирования исключительных ситуаций, возникновение которых фиксируется с помощью шести бит регистра `swr`. Если какие-то биты исключений в регистре `cwr` установлены в единицу, то это означает, что соответствующие исключения будут обрабатываться самим сопроцессором. Если для какого-либо исключения в соответствующем бите масок исключений регистра `cwr` содержится нулевое значение, то при возникновении исключения этого типа будет возбуждено прерывание 16 (10h). Операционная система должна содержать (или программист должен написать) обработчик этого прерывания. Он должен выяснить причину прерывания, после чего, если это необходимо, исправить ее, а также выполнить другие действия. Более подробно этот вопрос обсуждается в конце урока.

Регистр управления `cwr`

| | <code>rc</code> | <code>pc</code> | | | <code>p</code> | <code>u</code> | <code>o</code> | <code>z</code> | <code>d</code> | <code>i</code> |
|----|-----------------|-----------------|---|---|----------------|----------------|----------------|----------------|----------------|----------------|
| 15 | 13 11 | 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 0 |

Рис. 19.4. Формат регистра управления сопроцессором `cwr`

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

- `pc=00` — длина мантиссы 24 бита;
- `pc=10` — длина мантиссы 53 бита;
- `pc=11` — длина мантиссы 64 бита.

По умолчанию устанавливается значение поля `pc=11`.

Поле управления округлением `rc` позволяет управлять процессом округления чисел в процессе работы сопроцессора. Необходимость операции округления может появиться в ситуации, когда после выполнения очередной команды со-

процессора получается не представимый результат, например, периодическая дробь 3,333... Установив одно из значений в поле `r c`, можно выполнить округление в необходимую сторону. Для того чтобы выяснить характер округления, введем обозначения:

- `m` – значение в `st(0)` или результат работы некоторой команды, который не может быть точно представлен и поэтому должен быть округлен;
- `a` и `b` – наиболее близкие значения к значению `m`, которые могут быть представлены в регистре `st(0)` сопроцессора, причем выполняется условие `a < m < b`.

Ниже приведены значения поля `r c` и описан соответствующий им характер округления:

- 00 – значение `m` округляется к ближайшему числу `a` или `b`;
- 01 – значение `m` округляется в меньшую сторону, то есть `m=a`;
- 10 – значение `m` округляется в большую сторону, то есть `m=b`;
- 11 – производится отбрасывание дробной части `m`. Используется для приведения значения к форме, которая может использоваться в операциях целочисленной арифметики.

Регистр тегов `twr`

Регистр тегов `twr` представляет собой совокупность двухбитовых полей. Каждое двухбитовое поле соответствует определенному физическому регистру стека (см. рис. 19.1) и характеризует его текущее состояние. Изменение состояния любого регистра стека отражается на содержимом соответствующего этому регистру поля регистра тега. Возможны следующие значения в полях регистра тега:

- 00 – регистр стека сопроцессора занят допустимым ненулевым значением;
- 01 – регистр стека сопроцессора содержит нулевое значение;
- 10 – регистр стека сопроцессора содержит одно из специальных численных значений (см. ниже), за исключением нуля;
- 11 – регистр пуст и в него можно производить запись. Нужно отметить, что это значение в одном из двухбитовых полей регистра тегов не означает, что все биты соответствующего регистра стека должны быть обязательно нулевыми.

Мы не раз уже отмечали, что при написании программы разработчик манипулирует не абсолютными, а относительными номерами регистров стека. По этой причине у него могут возникнуть трудности при попытке интерпретации содержимого регистра тегов `twr`, с соответствующими физическими регистрами стека. В качестве связующего звена необходимо привлекать информацию из поля `top` регистра `swr`.

Форматы данных

Сопроцессор расширяет номенклатуру форматов данных, с которыми работает основной процессор. В этом нет ничего удивительного, так как формат данных любого устройства в существенной мере отражает специфику его работы. Сопро-

цессор специально разрабатывался для вычислений с плавающей точкой. Но сопроцессор может работать и с целыми числами, хотя и менее эффективно. Перечислим форматы данных, с которыми работает сопроцессор:

- двоичные целые числа в трех форматах — 16, 32 и 64 бита;
- упакованные целые десятичные (BCD) числа — максимальная длина 18 упакованных десятичных цифр (9 байт);
- вещественные числа в трех форматах — коротком (32 бита), длинном (64 бита), расширенном (80 бит).

Кроме этих основных форматов, сопроцессор поддерживает специальные численные значения, к которым относятся:

- денормализованные вещественные числа — это числа меньшие минимального нормализованного числа (см. ниже) для каждого вещественного формата, поддерживаемого сопроцессором;
- нуль;
- положительные и отрицательные значения бесконечность;
- нечисла;
- неопределенности и неподдерживаемые форматы.

Рассмотрим более подробно основные форматы данных, поддерживаемые сопроцессором. Важно отметить, что в самом сопроцессоре числа в этих форматах имеют одинаковое внутреннее представление — в виде расширенного формата вещественного числа. Это один из форматов представления вещественных чисел, который точно соответствует формату регистров $r0 \dots r7$ стека сопроцессора (рис. 19.1). Таким образом, даже если вы используете команды сопроцессора с целочисленными операндами, то после загрузки в сопроцессор операндов целого типа они автоматически преобразуются в формат расширенного вещественного числа.

Двоичные целые числа

Сопроцессор работает с тремя типами целых чисел (рис. 19.5). В табл. 19.1 представлен формат целых чисел, их размерность и диапазон значений.

Таблица 19.1. Форматы целых чисел сопроцессора

| Формат | Размер, бит | Диапазон значений |
|----------------|-------------|---|
| Целое слово | 16 | -32 768...+32 767 |
| Короткое целое | 32 | $-2 \times 10^9 \dots +2 \times 10^9$ |
| Длинное целое | 64 | $-9 \times 10^{18} \dots +9 \times 10^{18}$ |

Выбирая формат данных, с которыми будет работать ваша программа, помните, что сопроцессор поддерживает операции с целыми числами, но работа с ними осуществляется неэффективно. Причина в том, что обработка сопроцессором целочисленных данных будет замедлена из-за необходимости выполнения дополнительного преобразования целых чисел в их внутреннее представление в виде эквивалентного вещественного числа расширенного формата.

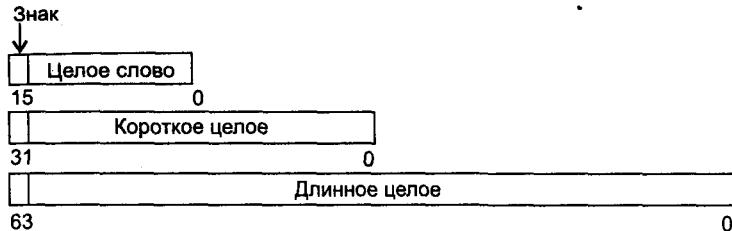


Рис. 19.5. Форматы целых чисел сопроцессора

В программе целые двоичные числа описываются обычным способом — с использованием директив dw, dd и dq. Например, целое число 5 может быть описано следующим образом:

```
ch_dw    5      ;представление в памяти: ch_dw=05 00
ch_dd    5      ;представление в памяти: ch_dw=05 00 00 00
ch_dq    5      ;представление в памяти: ch_dw=05 00 00 00 00 00 00 00
```

Работать с целыми числами может далеко не всякая команда сопроцессора. Подробную информацию о командах сопроцессора можно найти в Справочнике.

Упакованные целые десятичные (BCD) числа

Сопроцессор использует один формат упакованных десятичных чисел (рис. 19.6). Как вы помните, для описания упакованного десятичного числа используется директива dt (приложение 6 и урок 8). Данная директива позволяет описать 20 цифр в упакованном десятичном числе (по две в каждом байте). Из-за того что максимальная длина упакованного десятичного числа в сопроцессоре составляет только 9 байт, в регистры r0...r7 можно поместить только 18 упакованных десятичных цифр. Старший десятый байт игнорируется. Самый старший бит этого байта используется для хранения знака числа.

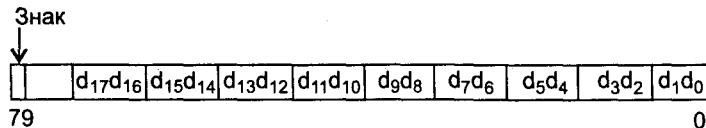


Рис. 19.6. Формат десятичного числа сопроцессора

Упакованные десятичные числа также представляются в стеке сопроцессора в расширенном формате. Упакованные десятичные числа в программе описываются директивой dt. Например, целое число 5365904 в формате упакованного десятичного числа может быть описано следующим образом:

```
ch_dt    dt    5365904
;представление в памяти: ch_dt=05 59 36 05 00 00 00 00 00 00
```

Нужно отметить, что сопроцессор имеет для работы с упакованными десятичными числами всего две команды — сохранения и загрузки.

Вещественные числа

Основной тип данных, с которыми работает сопроцессор — вещественный. Данные этого типа описываются тремя форматами: коротким, длинным и расширенным (рис. 19.7).

| Знак | Характеристика q | Мантисса (M) | | |
|------|------------------|--------------|---|--------------------|
| ↓ | 31 | 24 23 | 0 | Короткий формат |
| | 63 | 53 52 | 0 | Длинный формат |
| | 79 | 64 63 | 0 | Расширенный формат |

Рис. 19.7. Форматы вещественных чисел сопроцессора

Для представления вещественного числа используется формула (19.1):

$$A = (\pm M)^{\pm p}, \quad (19.1)$$

где M — мантисса числа A . Мантисса должна удовлетворять условию $|M| < 1$;

N — основание системы счисления, представленное целым положительным числом;

p — порядок числа, показывающий истинное положение точки в разрядах мантиссы (по этой причине вещественные числа имеют еще название чисел с плавающей точкой, так как ее положение в разрядах мантиссы зависит от значения порядка).

Для удобства обработки в компьютере чисел с плавающей точкой, архитектурой компьютера на компоненты формулы (19.1) накладываются некоторые ограничения. Для сопроцессоров, применяющихся в архитектуре Intel, эти условия и ограничения заключаются в следующем:

- Основание системы счисления $N=2$.
- Мантисса M должна быть представлена в *нормализованном* виде. Нормализация может отличаться для разных типов процессоров. Для ЕС ЭВМ, например, мантисса нормализованного числа должна удовлетворять условию: $1/N < |M| < 1$. Это означает, что старший бит представления должен быть единичным. Для случая, когда $N=2$ это соответствует отношению $1/2 \leq |M| < 1$ или в двоичном виде $0,10...00 \leq |M| < 0,11...11$, то есть первая цифра после запятой должна быть значащей (единицей), а порядок p , соответственно, таким, чтобы это условие выполнялось. Для архитектуры микропроцессора Intel нормализованным является число несколько иного вида:

$$A = (-1)^s N^p M \quad (19.2)$$

где S — значение знакового разряда:

0 — число больше нуля;

1 — число меньше нуля;

p — порядок числа. Его значение аналогично значению порядка p в формуле (19.1).

В этой формуле знак имеют и порядок вещественного числа, и его мантисса. На рис. 19.7 видно, что формат хранения вещественного числа в памяти имеет только поле для знака мантиссы. А где же хранится знак порядка?

В сопроцессоре Intel на аппаратном уровне принято соглашение, что порядок p определяется в формате вещественного числа особым значением, называемым *характеристикой* q . Величина q связана с порядком p посредством формулы (19.3) и представляет собой некоторую константу. Условно назовем ее *фиксированным смещением*.

$$q = p + \text{фиксированное смещение} \quad (19.3)$$

Для каждого из трех возможных форматов вещественных чисел смещение q имеет разное, но фиксированное для конкретного формата значение, которое зависит от количества разрядов, отводимых под характеристику (табл. 19.2).

Таблица 19.2. Формат вещественных чисел

| Формат | Короткий | Длинный | Расширенный |
|----------------------------------|---------------------------|-----------------------------|-------------------------------|
| Длина числа (биты) | 32 | 64 | 80 |
| Размерность мантиссы M | 24 | 53 | 64 |
| Диапазон значений | $10^{-38} \dots 10^{+38}$ | $10^{-308} \dots 10^{+308}$ | $10^{-4932} \dots 10^{+4932}$ |
| Размерность характеристики q | 8 | 11 | 15 |
| Значение фиксированного смещения | +127 | +1023 | +16383 |
| Диапазон характеристик q | 0...255 | 0...2047 | 0...32767 |
| Диапазон порядков p | -126...+127 | -1022...+1023 | -16382...+16383 |

В табл. 19.2 показаны диапазоны значений характеристик q и соответствующих им истинных порядков p вещественных чисел. Отметим, что нулевому порядку вещественного числа в коротком формате соответствует значение характеристики равное 127, которому в двоичном представлении соответствует значение 01 11 11 11. Отрицательному порядку p , например, -1, будет соответствовать характеристика $q = -1 + 127 = 126$. В двоичном виде ей соответствует значение 01 11 11 10. Положительному порядку p , например, +1, будет соответствовать характеристика $q = 1 + 127 = 128$, в двоичном виде ей соответствует значение 10 00 00 00. То есть, все положительные порядки имеют в двоичном представлении характеристики старший бит равный единице, а отрицательные порядки — нет. Не кажется ли вам, что мы нашли место, где спрятан знак порядка? В старшем бите характеристики. Теперь вам должно быть понятно, откуда появились значения в двух последних строках табл. 19.2.

Так как нормализованное вещественное число всегда имеет целую единичную часть (исключая представление перечисленных выше специальных числовых значений), то при его представлении в памяти появляется возможность считать первый разряд вещественного числа единичным по умолчанию и учитывать его наличие только на аппаратном уровне. Это дает возможность увеличить диапазон представимых чисел, так как появляется лишний разряд, который можно использовать для представления мантиссы числа. Но это справедливо только для

короткого и длинного форматов вещественных чисел. Расширенный формат, как внутренний формат представления числа любого типа в сопроцессоре, содержит целую единичную часть вещественного в явном виде.

Как определить вещественное число или зарезервировать место для его размещения в программе на ассемблере?

Короткое вещественное число длиной в 32 разряда определяется директивой `dd`. При этом обязательным в записи числа является наличие десятичной точки, даже если оно не имеет дробной части. Для транслятора десятичная точка является указанием, что число нужно представить в виде числа с плавающей точкой в коротком формате (рис. 19.7). Это же касается длинного и расширенного форматов представления вещественных чисел, определяемых директивами `dq` и `dt`. Другой способ задания вещественного числа директивами `dd`, `dq` и `dt` — экспоненциальная форма с использованием символа «`e`». Вид вещественного числа в поле операндов директив `dd`, `dq` и `dt` можно представить синтаксической диаграммой (рис. 19.8).

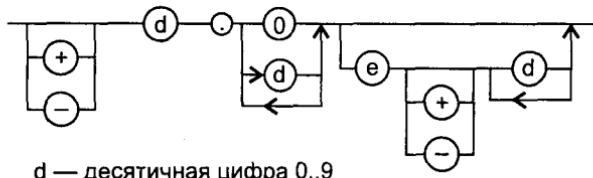


Рис. 19.8. Синтаксис вещественных чисел в директивах `dd`, `dq` и `dt`

Чтобы окончательно разобраться с тонкостями представления вещественных чисел различных форматов в памяти, рассмотрим несколько примеров.

Пример 19.1. Определение вещественного числа короткого формата

Определим в программе вещественное число 45,56 в коротком формате:

`dd 45.56`

или

`dd 45.56e0`

или

`dd 0.4556e2`

В памяти это число будет выглядеть так:

`71 3d 36 42`

Учитывая, что в архитектуре Intel принят перевернутый порядок следования байт в памяти в соответствии с принципом «младший байт по младшему адресу», истинное представление числа 45,56 будет следующим:

`42 36 3d 71`

В двоичном представлении в памяти это будет иметь вид, как на рис. 19.9.

На рис. 19.9 видно, что старшая единица мантиссы, при представлении в памяти, отсутствует.

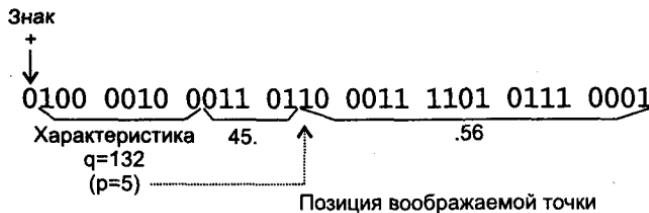


Рис. 19.9. Двоичное представление в памяти вещественного числа в директиве dd

Пример 19.2. Определение вещественного числа длинного формата

Определим в программе вещественное число 45,56 в длинном формате:

`dq 45.56`

или

`dq 45.56e0`

В памяти это число будет выглядеть так:

`47 e1 7a 14 ae c7 46 40`

Перевернув его, получим истинное значение:

`40 46 c7 ae 14 7a e1 47`

Разберите его по компонентам вещественного числа самостоятельно.

Пример 19.3. Определение вещественного числа расширенного формата

Определим в программе вещественное число 45,56 в расширенном формате представления:

`dt 45.56`

В памяти это число будет выглядеть так:

`71 3d 0a d7 a3 70 3d b6 04 40`

Перевернув его, получим истинное значение в памяти:

`40 04 b6 3d 70 a3 d7 0a 3d 71`

А вот его двоичное представление полезно рассмотреть подробнее (рис. 19.10):

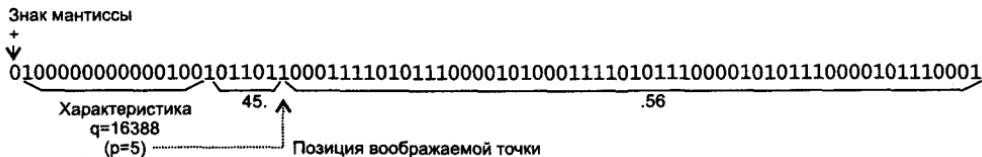


Рис. 19.10. Двоичное представление в памяти вещественного числа в директиве dt

0100 0000 0000 0100 1011 0110 0011 1101 0111 0000 1010
0011 1101 0111 0000 1010 1110 0001 0111 0001

Данное число имеет следующее назначение битов:

- 0 – знак «+»;
- 100 0000 0000 0100 – характеристика $q=1,6388$;

- 1011 01 – целая часть числа (45);
- 10 0011 1101 0111 0000 1010 0011 1101 0111 0000
1010 1110 0001 0111 0001 – дробная часть числа (0,56).

Как видно, в мантиссе явно присутствует старшая единица, чего не было в коротком и длинном форматах представления вещественного числа.

Дальнейшее обсуждение требует четкого понимания того, каким образом из дробного десятичного числа получается его расширенное вещественное представление. Рассмотрим этот процесс по шагам.

1. Переведем десятичную дробь 45,56 в двоичное представление. Подробно алгоритм такого перевода описан на уроке 6. В результате получим эквивалентное двоичное представление:

$$45,56_{10} = 101101.1000111101011100001010001 \\ 111010111000010101110000101110001$$

2. Нормализуем число. Для этого переносим точку влево, до тех пор, пока в целой части числа не останется одна двоичная единица. Число переносов влево (или вправо, если десятичное число было меньше единицы) будет являться порядком числа. Но будьте внимательны – в поле характеристики заносится смещение значение порядка (табл. 19.2). Таким образом, после перемещения точки получаем значение порядка равное 5. Соответственно, характеристика будет выглядеть так: $5+16383=16388_{10}=1000000000000100_2$.

Сформированный результат в виде вещественного числа в расширенном формате состоит из трех компонент:

- знака – 0;
- характеристики – 100000000000100;
- мантиссы – 1011 0110 0011 1101 0111 0000 1010 0011
1101 0111 0000 1010 1110 0001 0111 0001

Ниже вы научитесь пользоваться отладчиком для работы с сопроцессором и получите возможность просматривать содержимое регистров стека. При этом содержимое регистров стека будет изображаться в шестнадцатеричном виде: 40 04 b6 3d 70 a3 d7 0a 3d 71. Видно, что он полностью совпадает с приведенным выше представлением числа в памяти, если оно описано в директиве dt.

В качестве итога еще раз подчеркнем, что расширенный формат представления вещественного числа является единственным форматом представления чисел в регистрах сопроцессора. Само преобразование производится автоматически при загрузке числа в стек сопроцессора. Исключение составляет расширенный формат.

Специальные численные значения

Несмотря на большой диапазон вещественных значений, представимых в регистрах стека сопроцессора, понятно, что бесконечное количество их значений находится за рамками этого диапазона. Для того чтобы иметь возможность реагировать на вычислительные ситуации, в которых возникают такие значения, в сопроцессоре предусмотрены специальные комбинации бит, называемые *специальными численными значениями*. При необходимости, программист может сам

кодировать специальные численные значения. Это возможно потому, что вещественные числа, описанные директивой `d t`, соответствующие команды сопроцессора загружают без всяких преобразований.

Денормализованные вещественные числа

Денормализованные вещественное числа — это числа, которые меньше минимального нормализованного числа для каждого вещественного формата. Поясним природу денормализованных чисел с использованием числовой шкалы. Например, для вещественного числа в расширенном формате диапазон представимых значений в сопроцессоре показан на числовой шкале рис. 19.11.

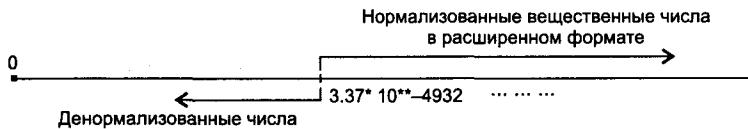


Рис. 19.11. Положение денормализованных вещественных чисел на числовой шкале

Как нам уже известно, сопроцессор хранит числа в нормализованном виде. По мере приближения чисел к нулю, ему все труднее «вытягивать» их значения к нормализованному виду, то есть к такому виду, чтобы первой значащей цифрой мантиссы была единица. Размерность разрядной сетки, отведенной в форматах вещественных чисел сопроцессора, для представления характеристики не безгранична. Поэтому при определенных значениях числа в расширенном формате значение характеристики становится равным нулю (рис. 19.11). Но на самом деле, число отлично от нуля, так как это все же не настоящий численный нуль. Таким образом, между истинным нулем и минимально представимым нормализованным числом есть еще бесконечное количество очень маленьких чисел. Это и есть так называемые денормализованные числа. Они имеют нулевой порядок и ненулевую мантиссу. Диапазон представимых в сопроцессоре денормализованных чисел не ограничен, так как количество разрядов мантиссы ограничено (рис. 19.12):

Минимальное положительное денормализованное число:

| | | | | |
|----|--------|-------|-------|-----|
| 0 | 00..00 | 00000 | | 001 |
| 79 | 78 | 64 | 63 | 0 |

Минимальное отрицательное денормализованное число:

| | | | | |
|----|--------|-------|-------|-----|
| 1 | 00..00 | 00000 | | 001 |
| 79 | 78 | 64 | 63 | 0 |

Максимальное положительное денормализованное число:

| | | | | |
|----|--------|-------|-------|-----|
| 0 | 00..00 | 11111 | | 111 |
| 79 | 78 | 64 | 63 | 0 |

Максимальное отрицательное денормализованное число:

| | | | | |
|----|--------|------|-------|-----|
| 1 | 00..00 | 1111 | | 111 |
| 79 | 78 | 64 | 63 | 0 |

Рис. 19.12. Диапазон представимых в сопроцессоре денормализованных чисел

Вопрос о том, каким образом сопроцессор реагирует на появление денормализованных значений, будет рассмотрен в конце урока. При формировании денорма-

лизованного значения в некотором регистре стека, в соответствующем этому регистру теге регистра `twr` формируется специальное значение (10).

Нуль

Значение **нуля** также относят к специальным численным значениям. Это делается из-за того, что это значение особо выделяется среди корректных вещественных значений, формируемых как результат работы некоторой команды. Более того, нуль может формироваться как реакция сопроцессора на определенную вычислительную ситуацию.

Значение истинного нуля может иметь знак (рис. 19.13), что, впрочем, не влияет на его восприятие командами сопроцессора. Если необходимо определить знак нуля, то используйте команду `fchm`. В результате работы этой команды в бит с 1 регистра `swr` заносится знак операнда. При загрузке нуля в регистр стека, в соответствующем теге регистра `twr` формируется специальное значение (01).

| | | | | |
|----|--------|-------|-------|-----|
| 0 | 00..00 | 0000 | | 000 |
| 79 | 78 | 64 63 | | 0 |
| 1 | 00..00 | 0000 | | 000 |
| 79 | 78 | 64 63 | | 0 |

Рис. 19.13. Представление нуля в регистре стека сопроцессора

Значение нуля может быть сформировано в результате возникновения ситуации антипереполнения (см. ниже), а также при работе команд с нулевыми operandами.

Значение бесконечность

Сопроцессор имеет средства для представления значения бесконечность. Это значение формируется с помощью специальных битовых значений. Формат регистра стека сопроцессора, содержащего значение бесконечность, приведен на рис. 19.14.

| | | | | |
|----|--------|-------|-------|-----|
| 0 | 11..11 | 10000 | | 000 |
| 79 | 78 | 64 63 | | 0 |
| 1 | 11..11 | 10000 | | 000 |
| 79 | 78 | 64 63 | | 0 |

Рис. 19.14. Представление значения бесконечность в регистре стека сопроцессора

Рисунок 19.14 демонстрирует, что значение бесконечность может иметь знак, при этом значения мантиссы и характеристики фиксированы. Именно в этом заключается отличие значения бесконечность от остальных специальных значений.

Среди причин, приводящих к формированию значения бесконечность, можно выделить переполнение и деление на нуль. При формировании значения бесконечность в некотором регистре стека, в соответствующем теге регистра `twr` формируется специальное значение (10).

Нечисла

К нечислам относятся такие битовые последовательности в регистре стека сопроцессора, которые не совпадают ни с одним из рассмотренных выше форматов значений. Нечисло должно иметь единичную мантиссу и любую мантиссу, кроме 100...00, которая зарезервирована для значения бесконечность. Различают два типа нечисел:

- SNAN (Signaling Non a Number) – сигнальные нечисла;
- QNaN (Quiet Non A Number) – спокойные (тихие) нечисла.

Сигнальное нечисло – битовое значение с единичным значением поля характеристики и мантиссы, первый бит которой, следующий за первым единичным значащим битом мантиссы, равен нулю (рис. 19.15, а). Сопроцессор реагирует на появление этого числа в регистре стека возбуждением исключения недействительная операция. Программисты могут формировать эти числа в регистре стека сопроцессора преднамеренно, например, для того чтобы искусственно возбудить в нужной ситуации указанное исключение. Очевидно, что именно по этой причине данные числа называются сигнальными. Если снять маску у флага недействительная операция в регистре *cwr*, то будет вызван обработчик, который выполнит заданные программистом действия.

| | | | | |
|---|------------|-------|---------|-----|
| a | x 11..11 | 10xxx | | xxx |
| | 79 78 | 64 63 | | 0 |
| b | x 11..11 | 11xxx | | xxx |
| | 79 78 | 64 63 | | 0 |
| c | 1 11..11 | 11000 | | 000 |
| | 79 78 | 64 63 | | 0 |

Рис. 19.15. Представление нечисел в регистре стека сопроцессора

Спокойное нечисло – битовое значение с единичным значением полей характеристики и мантиссой, первые два бита которой равны единице (рис. 19.15, б).

Сопроцессор самостоятельно не формирует сигнальных чисел, но в качестве реакции на определенные исключения он может формировать спокойные нечисла, например, нечисло *вещественная неопределенность*. Его значение показано на рис. 19.15, в. Вещественная неопределенность формируется как маскированная реакция сопроцессора на исключение недействительная операция (см. раздел «Исключения сопроцессора и их обработка» в конце урока). Другие спокойные нечисла могут формироваться после выполнения команд, в которых хотя бы один из операндов был спокойным нечислом. Это может породить цепную реакцию, которая приведет к ошибочному результату. Поэтому в процессе вычислений рекомендуется периодически контролировать результаты исполнения команд на предмет появления спокойных нечисел.

При формировании нечисла в некотором регистре стека, в соответствующем теге регистра *twr* формируется специальное значение (10).

Неподдерживаемые форматы

Необходимо иметь в виду, что существует достаточно много битовых наборов, которые можно представить в расширенном формате вещественного числа.

Для большинства их значений формируется исключение недействительная операция.

Система команд сопроцессора

Система команд сопроцессора включает в себя около 80 машинных команд. Рассмотрим их классификацию (рис. 19.16).

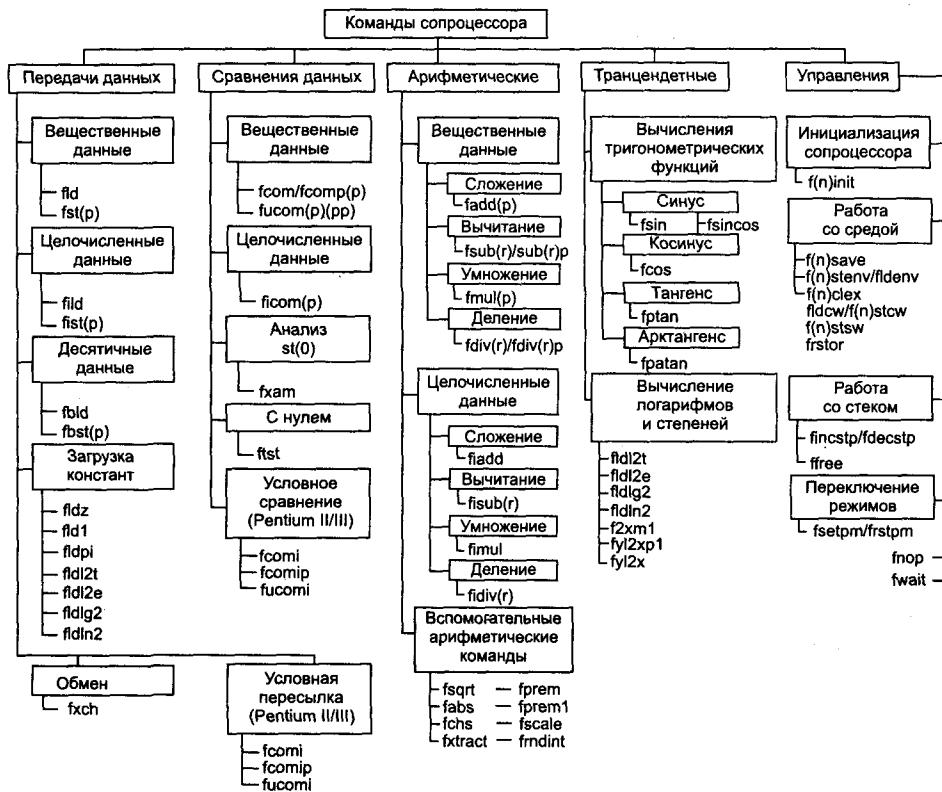


Рис. 19.16. Функциональная классификация команд сопроцессора

Мнемоническое обозначение команд сопроцессора характеризует особенности их работы и в связи с этим может представлять определенный интерес. Поэтому коротко рассмотрим основные моменты образования названий команд:

- все мнемонические обозначения начинаются с символа f (float);
- вторая буква мнемонического обозначения определяет тип операнда в памяти, с которым работает команда:
 - i — целое двоичное число;
 - b — целое десятичное число;
 - отсутствие буквы — вещественное число;

- последняя буква мнемонического обозначения команды `r` означает, что последним действием команды обязательно является извлечение операнда из стека;
- последняя или предпоследняя буква `r` (`reversed`) означает реверсивное следование операндов при выполнении команд вычитания и деления, так как для них важен порядок следования операндов.

Полезной может оказаться и информация о машинных форматах команд сопроцессора. Если давать им общую характеристику, то важно отметить, что в целом, система команд сопроцессора отличается большой гибкостью в выборе вариантов задания команд, реализующих определенную операцию, и их operandов. Минимальная длина команды сопроцессора — 2 байта. При обсуждении команд в тексте данного урока будет приводиться лишь их схема. Детально машинное представление команд сопроцессора и сами команды описаны в Справочнике.

Методика написания программ для сопроцессора имеет свои особенности. Главная причина здесь в стековой организации сопроцессора. Для того чтобы написать программу для вычисления некоторого выражения, его необходимо предварительно преобразовать в удобный для программирования сопроцессора вид. Процесс преобразования напоминает подготовку выражения для метода трансляции, основанного на использовании *обратной польской записи* (ПОЛИЗ). Рассмотрим суть ПОЛИЗ на примере вычисления выражения:

$$a+b*c-d/(a+b).$$

Графически это выражение представляется в виде дерева (рис. 19.17).

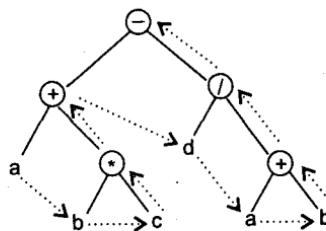


Рис. 19.17. Представление выражения в виде дерева

Листья ветвей дерева соответствуют operandам, а узлы — операциям. Пусть началом обхода будет лист самой левой ветви дерева. Тогда для получения обратной польской записи выражения необходимо двигаться по дереву слева направо, при этом узлы должны просматриваться только после обхода всех исходящих из него ветвей. В результате выражение будет выглядеть так:

$$abc*+dab+-.$$

Другое название такой формы записи — *постфиксная запись*. Такое название означает, что знак операции записывается после operandов, участвующих в операции. Использование постфиксной записи позволяет вычислять выражения за один проход с учетом приоритета арифметических операций.

Алгоритм вычисления выражений в постфиксной записи имеет следующий вид:

1. Выбрать очередной символ записи выражения в форме ПОЛИЗ.
2. Если очередной выбранный символ — operand, то поместить его в стек, после чего перейти к шагу 1.

3. Если очередной выбранный символ — знак операции, то выполнить ее над одним или двумя operandами в вершине стека. Результат операции необходимо поместить обратно в вершину стека.
4. Если в исходной записи выражения в форме ПОЛИЗ еще остались символы, то перейти к шагу 1, иначе в вершине стека находится результат вычисления выражения.

При разработке программ необходимо учитывать следующие факторы:

- ограниченность глубины стека сопроцессора;
- несовпадение форматов operandов;
- отсутствие поддержки на уровне команд сопроцессора некоторых операций, таких как возведение в степень, вычисление тригонометрических функций.

Для решения этих проблем вам придется разрабатывать дополнительный программный код, отклоняясь от последовательной классической обработки арифметических выражений, представленных в виде ПОЛИЗ. Однако стержнем программы будет строка в виде ПОЛИЗ.

На диске в подкаталоге данного урока находятся файлы программы, с помощью которой вы можете облегчить себе задачу преобразования простого арифметического выражения в форму ПОЛИЗ. Она воспринимает ограниченное количество операций, набор которых вы можете расширить, дополнив конфигурационный файл `polis.cfg` и исходный текст программы.

Здесь возникает интересная алгоритмическая задача. В чем ее суть? Для тех операций, которые реализованы на уровне команд сопроцессора, особых проблем нет. И наоборот, для выражений, содержащих операции, не реализованные на уровне команд сопроцессора, возникает проблема преобразования исходной строки в форму ПОЛИЗ. В качестве примера можно привести такую распространенную операцию, как вычисление степени. Соответствующую программу мы рассмотрим ниже в тексте урока. Она содержит ряд операций, которые должны быть выполнены для вычисления степени. Таким образом, преобразованное в форму ПОЛИЗ выражение будет содержать ряд операций, которые отсутствуют в исходной записи этого выражения. Это означает, что при формировании строки ПОЛИЗ программа `r_main` должна выполнять неявную подстановку дополнительных operandов и операций в соответствии с некоторой формулой приведения.

Команды передачи данных

Группа команд передачи данных предназначена для организации обмена между регистрами стека, вершиной стека сопроцессора и ячейками оперативной памяти. Команды этой группы имеют такое же значение для процесса программирования сопроцессора, как и команда `mov` основного процессора. С помощью этих команд осуществляются все перемещения значений operandов в сопроцессор и из него. По этой причине для каждого из трех типов данных, с которыми может работать сопроцессор, существует своя подгруппа команд передачи данных. Собственно на этом уровне все его умения по работе с различными форматами данных и заканчиваются. Главной функцией всех команд загрузки данных в сопроцессор является преобразование их к единому представлению в виде вещественного числа расширенного формата. Это же касается и обратной операции — сохранения в памяти данных из сопроцессора.

Команды передачи данных можно разделить на следующие группы:

- команды передачи данных в вещественном формате;
- команды передачи данных в целочисленном формате;
- команды передачи данных в десятичном формате.

Команды передачи данных в вещественном формате:

f1d источник — загрузка вещественного числа из области памяти в вершину стека сопроцессора;

fst приемник — сохранение вещественного числа из вершины стека сопроцессора в память. Как следует из анализа мнемокода команды (отсутствует символ **r**), сохранение числа в памяти не сопровождается выталкиванием его из стека, то есть текущая вершина стека сопроцессора не изменяется (поле **top** не изменяется);

fstp приемник — сохранение вещественного числа из вершины стека сопроцессора в память. В отличие от предыдущей команды, в конце мнемонического обозначения данной команды присутствует символ **r**, что означает выталкивание вещественного числа из стека после его сохранения в памяти. Команда изменяет поле **top**, увеличивая его на единицу. Вследствие этого, вершиной стека становится следующий, больший по своему физическому номеру, регистр стека сопроцессора.

Команды передачи данных в целочисленном формате:

fild источник — загрузка целого числа из памяти в вершину стека сопроцессора;

fist приемник — сохранение целого числа из вершины стека сопроцессора в память. Сохранение целого числа в памяти не сопровождается выталкиванием его из стека, то есть текущая вершина стека сопроцессора не изменяется;

fistp приемник — сохранение целого числа из вершины стека в память. Аналогично сказанному выше о команде **fstp**, последним действием команды является выталкивание числа из стека с одновременным преобразованием его в целое значение.

Команды передачи данных в десятичном формате:

fbl d источник — загрузка десятичного числа из памяти в вершину стека сопроцессора;

fbstp приемник — сохранение десятичного числа из вершины стека сопроцессора в области памяти. Значение выталкивается из стека после преобразования его в формат десятичного числа. Заметьте, что для десятичных чисел нет команды сохранения значения в памяти без выталкивания из стека.

К группе команд передачи данных можно отнести команду обмена вершины регистрастрового стека **st(0)** с любым другим регистром стека:

fxch st(i) — обмен значений между текущей вершиной стека и регистром стека сопроцессора **st(i)**.

Действие команд загрузки **f1d**, **fild** и **fbl d** можно сравнить с командой **push** основного процессора. Аналогично ей (**push** уменьшает значение в регистре **sp**), команды загрузки сопроцессора перед сохранением значения в регистровом стеке сопроцессора вычитают из содержимого поля **top** регистра состояния **swr** единицу. Это означает, что вершиной стека становится регистр с физическим номером

на единицу меньше. При этом возможно переполнение стека. Так как стек сопроцессора состоит из ограниченного числа регистров, то в него может быть записано максимум восемь значений. Из-за кольцевой организации стека, девятое записываемое значение затрет первое. Программа должна иметь возможность обработать такую ситуацию. По этой причине почти все команды, помещающие свой операнд в стек сопроцессора, после уменьшения значения поля `top`, проверяет регистр — кандидат на новую вершину стека — на предмет его занятости. Для анализа этой и подобных ситуаций используется регистр `twr`, содержащий слово тегов (см. рис. 19.1). Наличие регистра тегов в архитектуре сопроцессора позволяет избежать разработки программистом сложной процедуры распознавания содержимого регистров сопроцессора и дает самому сопроцессору возможность фиксировать определенные ситуации, например, попытку чтения из пустого регистра или запись в непустой регистр. Возникновение таких ситуаций фиксируется в регистре состояния `swr` (см. рис. 19.2), предназначенному для сохранения общей информации о сопроцессоре. Используя специальные команды сопроцессора, можно извлечь или напротив записать в него информацию (см. раздел «Команды управления сопроцессором»).

В качестве примера определим несколько констант и выполним несколько манипуляций по пересылке их в регистры сопроцессора и обратно в оперативную память или другой регистр стека. При этом будем следить за состоянием стека сопроцессора.

У читателя видимо возникнет вопрос о том, нет ли каких-либо средств для наблюдения за состоянием регистров сопроцессора, аналогично тому, как это делалось при работе с основным процессором. Для него, как вы помните, мы использовали отладчик Turbo Debugger. Оказывается, что далеко ходить не нужно. Тот же Turbo Debugger предоставляет нам эту возможность. Чтобы не нарушать структуру общего изложения, материал по Turbo Debugger вынесен в конец занятия. Вы можете ознакомиться с ним сейчас, после чего вернуться обратно. Если же вы используете другой отладчик, то следует обратиться к его описанию с тем, чтобы поискать аналогичные возможности.

Ведите следующую программу. Выполните ее под управлением отладчика и проследите за тем, как меняется состояние регистров сопроцессора в окне Numeric processor (см. раздел «Использование отладчика» в конце урока).

Листинг 19.1. Исследование команд передачи данных

```
;-----fpu.asm-----
.586p
masm
model    use16 small
.stack   100h
.data    ;сегмент данных
ch_dt    dt    43567 ;ch_dt=00 00 00 00 00 00 00 04 35 67
x_dw     3     ;x=00 03
y_real   dq    34e7  ;y_real=41 b4 43 fd 00 00 00 00
ch_dt_st dt    0
x_st     dw    0
y_real_st dq    0
.code
main     proc   ;начало процедуры main
```

```

mov    ax, @data
mov    ds, ax
fbld  ch_dt ;st(0)=43567
fld   x     ;st(1)=43567, st(0)=3
fld   y_real;st(2)=43567, st(1)=3, st(0)=340000000
fxch  st(2) ;st(2)=340000000, st(1)=3, st(0)=43567
fbstp ch_dt_st ;st(1)=340000000, st(0)=3
;ch_dt_st=00 00 00 00 00 00 04 35 67
fistp x_st  ;st(0)=340000000, x_st=00 03
fstp  y_real_st ;y_real_st=41 b4 43 fd 00 00 00 00
exit:
    mov    ax, 4c00h
    int    21h
main  endp
end main

```

Исследование работы этой программы полезно вести с открытым окном Dump, так как при этом хорошо видны различия в представлении типов данных.

Команды загрузки констант

Основным назначением сопроцессора является поддержка вычислений с плавающей точкой. В математических вычислениях достаточно часто встречаются предопределенные константы. Сопроцессор хранит значения некоторых из них. Другая причина использования этих констант заключается в том, что для определения их в памяти (в расширенном формате) требуется 10 байт, а это для хранения, например, единицы, расточительно (сама команда загрузки константы, хранящейся в сопроцессоре, занимает два байта). В формате, отличном от расширенного, эти константы хранить не имеет смысла, так как теряется время на их преобразование в тот же расширенный формат. Для каждой предопределенной константы существует своя специальная команда, которая производит загрузку ее в вершину регистрового стека сопроцессора:

`f1dz` – загрузка нуля в вершину стека сопроцессора;

`f1d1` – загрузка единицы в вершину стека сопроцессора;

`f1dp1` – загрузка числа π в вершину стека сопроцессора;

`f1d12t` – загрузка двоичного логарифма десяти в вершину стека сопроцессора;

`f1d12e` – загрузка двоичного логарифма числа e в вершину стека сопроцессора;

`f1d1g2` – загрузка десятичного логарифма двух в вершину стека сопроцессора;

`f1d1n2` – загрузка натурального логарифма двух в вершину стека сопроцессора.

Команды сравнения данных

Команды данной группы выполняют сравнение значений числа в вершине стека и операнда, указанного в команде:

`fcom [операнд_в_памяти]` – команда без operandов сравнивает два значения: одно находится в регистре `st(0)`, другое в регистре `st(1)`. Если указан `[операнд_в_памяти]`, то сравнивается значение в регистре `st(0)` стека сопроцессора со значением в памяти;

fcom *операнд* — команда сравнивает значение в вершине стека сопроцессора *st(0)* со значением *операнда*, который находится в регистре или в памяти. Последним действием команды является выталкивание значения из *st(0)*;

fcompp *операнд* — команда аналогична по действию **fcom** без operandов, но последним ее действием является выталкивание из стека значений обоих регистров *st(0)* и *st(1)*;

ficom *операнд_в_памяти* — команда сравнивает значение в вершине стека сопроцессора *st(0)* с целым операндом в памяти. Длина целого операнда 16 или 32 бита, то есть целое слово и короткое целое (табл. 19.1);

ficomr *операнд* — команда сравнивает значение в вершине стека сопроцессора *st(0)* с целым операндом в памяти. После сравнения и установки бит с 3–с 0 команда выталкивает значение из *st(0)*. Длина целого операнда 16 или 32 бита, то есть целое слово и короткое целое (см. табл. 19.1).

Следующая команда — сравнение значения в вершине стека с нулем:

ftst — команда не имеет operandов и сравнивает значения в *st(0)* со значением 00.

Предыдущие команды сравнения работают корректно, если operandы в них являются целыми или вещественными числами. Когда один из operandов оказывается нечислом, то фиксируется исключение недействительная ситуация, а коды условия с3-с0 соответствуют исключительной ситуации несравнимые или неупорядоченные operandы. Само же действие сравнения не производится. Микропроцессор предоставляет три команды, которые позволяют все же произвести сравнение таких operandов, но как вещественных чисел без учета их порядков:

fucm *st(i)* — команда сравнивает значения (без учета их порядков) в регистрах стека сопроцессора *st(0)* и *st(i)*;

fucomp *st(i)* — команда сравнивает значения (без учета их порядков) в регистрах стека сопроцессора — в *st(0)* и *st(i)*. Последним действием команды является выталкивание значения из вершины стека;

fucompp *st(i)* — команда сравнивает значения (без учета их порядков) в регистрах стека сопроцессора — в *st(0)* и *st(i)*. Последние два действия команды одинаковы — выталкивание значения из вершины стека.

В результате работы команд сравнения в регистре состояния устанавливаются следующие значения бит кода условия с3, с2, с0:

- если *st(0)*>operand, то 000 (с3с2с0);
- если *st(0)*<operand, то 001 (с3с2с0);
- если *st(0)*=operand, то 100 (с3с2с0);
- если operandы неупорядочены, то 111 (с3с2с0).

Для того чтобы получить возможность реагировать на эти коды командами условного перехода основного процессора (вспомните, что они реагируют на флаги в *eFlags*), нужно как-то записать сформированные биты условия с3, с2, с0 в регистр *eFlags*. В системе команд сопроцессора существует команда **fstsw**, которая позволяет запомнить слово состояния сопроцессора в регистре *ax* или ячейке памяти. Далее значения нужных бит извлекаются и анализируются командами основного процессора. Например, запись старшего байта слова состояния, в котором находятся биты с0-с3, в младший байт регистра *eFlags/Flags* осуществляется коман-

дой sahf. Эта команда записывает содержимое ah в младший байт регистра eflags/flags. После этого бит с 0 записывается на место флага с f, с 2 — на место pf, с 3 — на место zf. Бит с 1 выпадает из общего правила, так как в регистре флагов на месте соответствующего ему бита находится единица. Анализ этого бита нужно проводить с помощью логических команд основного процессора. Зная все это, вам остается, исходя из особенностей своего алгоритма, применять те команды условного перехода, которые анализируют состояние указанных флагов.

В качестве иллюстрации рассмотрим программу разбиения массива вещественных чисел в формате двойного слова на два массива. В первый массив поместим все элементы, которые больше или равны нулю, а во второй — меньше нуля.

Листинг 19.2. Исследование команд сравнения данных

```
.586p
masm
model use16 small
.stack 100h
.data ;сегмент данных
;исходный массив
masdd -2.0, 45.7, -9.4, 7.3, 60.3, -58.44, 890e7, -98746e3
mas_h_0 dd 8 dup (0) ;массив значений больше либо равных 0
i_mas_h_0 dd 0 ;текущий индекс в mas_h_0
mas_l_0 dd 8 dup (0) ;массив значений меньших 0
i_mas_l_0 dd 0 ;текущий индекс в mas_l_0
.code
main proc ;начало процедуры main
    mov ax, @data
    mov ds, ax
    xor esi, esi
    mov cx, 8 ;счетчик циклов
    finit ;приведение сопроцессора в начальное состояние
    fldz ;загрузка нуля в st(0)
cyc1:
    fcom mas[esi*4] ;сравнение нуля в st(0) с очередным элементом массива mas
    fstsw ax ;сохранение swr в регистре ax
    sahf ;запись swr->ax-> регистр флагов
    jp error ;переход по «плохому» операнду в команде fcom
    jc hi_0 ;переход, если mas[esi*4]>= 0 (mas[esi*4]>=st(0))
;пересылка операнда mas[esi*4] меньшего 0 в массив mas_l_0
    mov eax, mas[esi*4]
    mov edi, i_mas_l_0
    mov mas_l_0[edi*4], eax
    inc i_mas_l_0
    jmp cyc1_bst
hi_0:
;пересылка операнда mas[esi*4] большего или равного 0 в массив mas_h_0
    mov eax, mas[esi*4]
    mov edi, i_mas_h_0
    mov mas_h_0[edi*4], eax
    inc i_mas_h_0
cyc1_bst:
    inc si
    loop cyc1
```

```
error:  
:здесь можно вывести сообщение об ошибке в задании операндов
```

```
exit:  
    mov    ax, 4c00h  
    int    21h  
main    endp  
end main
```

К группе команд сравнения данных логично отнести и команду `fxam`:

`fxam` — команда анализирует operand в вершине стека сопроцессора `st(0)` и формирует значение бит `c0`, `c1`, `c2`, `c3` в регистре состояния сопроцессора `swr`. По состоянию этих битов можно судить о:

- знаке мантиссы — знаковый бит операнда в `st(0)` заносится в бит `c0` регистра `swr`;
- корректности записи вещественного числа в `st(0)`. Идентифицируются пустой регистр, корректное вещественное число, нечисло и неизвестный формат;
- типе специального численного значения: бесконечность, нуль, денормализованный operand.

Подробное описание команды `fxam` можно найти в Справочнике.

Арифметические команды

Команды сопроцессора, входящие в данную группу, реализуют четыре основные арифметические операции — сложение, вычитание, умножение и деление. Имеется также несколько дополнительных команд, предназначенных для повышения эффективности использования основных арифметических команд. С точки зрения типов operandов, арифметические команды сопроцессора можно разделить на команды, работающие с вещественными и целыми числами. Рассмотрим их.

Целочисленные арифметические команды

Целочисленные арифметические команды предназначены для работы на тех участках вычислительных алгоритмов, где в качестве исходных данных используются целые числа в памяти в формате слово и короткое слово, имеющие раз мерность 16 и 32 бит.

При рассмотрении целочисленных арифметических команд, обратите внимание на большую гибкость задания operandов в этих командах.

`fiadd источник` — команда складывает значение `st(0)` и целочисленного *источника*, в качестве которого выступает 16- или 32-разрядный operand в памяти. Результат сложения запоминается в регистре стека сопроцессора `st(0)`;

`fisub источник` — команда вычитает значение целочисленного *источника* из `st(0)`. Результат вычитания запоминается в регистре стека сопроцессора `st(0)`. В качестве *источника* выступает 16- или 32-разрядный целочисленный operand в памяти;

`fimul источник` — команда умножает значение целочисленного *источника* на содержимое `st(0)`. Результат умножения запоминается в регистре стека

сопроцессора $st(0)$. В качестве *источника* выступает 16-ти или 32-разрядный целочисленный операнд в памяти;

fdiv источник — команда делит содержимое $st(0)$ на значение целочисленного *источника*. Результат деления запоминается в регистре стека сопроцессора $st(0)$. В качестве *источника* выступает 16- или 32-разрядный целочисленный операнд в памяти.

Для команд, реализующих арифметические действия деления и вычитания, важен порядок расположения операндов. По этой причине, система команд сопроцессора содержит соответствующие реверсивные команды, повышающие удобство программирования вычислительных алгоритмов. Чтобы отличить эти команды от обычных команд деления и вычитания, их мнемокоды оканчиваются символом *r*.

fisubr источник — команда вычитает значение $st(0)$ из целочисленного *источника*. Результат вычитания запоминается в регистре стека сопроцессора $st(0)$. В качестве *источника* выступает 16- или 32-разрядный целочисленный операнд в памяти;

fdivr источник — команда делит значение целочисленного *источника* на содержимое $st(0)$. Результат деления запоминается в регистре стека сопроцессора $st(0)$. В качестве *источника* выступает 16- или 32-разрядный целочисленный операнд в памяти.

Рассмотрим пример программы вычисления значения *u*:

$$u = \begin{cases} (x - y)/a, & \text{если } a \neq 0 \\ x \cdot y, & \text{если } a = 0 \end{cases}$$

Все переменные *x*, *y* и *a* целого типа в формате слова. Результат необходимо сохранить в ячейке памяти в формате десятичного числа.

Листинг 19.3. Исследование целочисленных арифметических команд

```
.586p
masm
model use16 small
.stack 100h
.data ;сегмент данных
;исходный массив
a dw 0
x dw 8
y dw 4
u dt 0
.code
main proc
    mov ax, @data
    mov ds, ax
    finit ;приведение сопроцессора в начальное состояние
    fld a ;загрузка значение а в st(0)
    fxam ;определяем тип а
    fstsw ax ;сохранение swr в регистре ax
    sahf ;запись swr->ax-> регистр флагов
    jc no_null
    jp no_null
```

```

jnz no_null
;вычисление формулы u=x+y:
    fild x
    fiadd y
    fbstp u
    jmp exit
no_null:
;вычисление формулы u=x-y/a:
    fild x
    fisub y
    fidiv a
    fbstp u
exit:
    mov ax, 4c00h
    int 21h
main endp
endmain

```

Вещественные арифметические команды

Схема расположения operandов вещественных команд традиционна для команд сопроцессора. Один из operandов располагается в вершине стека сопроцессора — регистре $st(0)$, куда после выполнения команды записывается и результат. Другой operand может быть расположен либо в памяти, либо в другом регистре стека сопроцессора. Допустимыми типами operandов в памяти являются все перечисленные выше вещественные форматы, за исключением расширенного формата.

В отличие от целочисленных арифметических команд, вещественные арифметические команды допускают большее разнообразие в сочетании местоположения operandов и самих команд для выполнения конкретного арифметического действия. Так, например, можно выделить три возможных варианта команды сложения:

fadd — команда складывает значения в $st(0)$ и $st(1)$. Результат сложения запоминается в регистре стека сопроцессора $st(0)$;

fadd источник — команда складывает значения $st(0)$ и *источника*, представляющего адрес ячейки памяти. Результат сложения запоминается в регистре стека сопроцессора $st(0)$;

fadd st(i), st — команда складывает значение в регистре стека сопроцессора $st(i)$ со значением в вершине стека $st(0)$. Результат сложения запоминается в регистре $st(i)$.

В дополнение к этим трем вариантам существует еще одна команда сложения, производящая дополнительное действие — удаление значения из стека:

faddp st(i), st — команда производит сложение вещественных operandов аналогично команде **fadd st(i), st**. Последним действием команды является выталкивание значения из вершины стека сопроцессора $st(0)$. Результат сложения остается в регистре $st(i-1)$.

Операция вычитание также выполняется с помощью большого набора команд:

fsub — команда вычитает значение в $st(1)$ из значения в $st(0)$. Результат вычитания запоминается в регистре стека сопроцессора $st(0)$;

fsub *источник* — команда вычитает значение *источника* из значения в $st(0)$. *Источник* представляет адрес ячейки памяти, содержащей допустимое вещественное число. Результат сложения запоминается в регистре стека сопроцессора $st(0)$;

fsub $st(i), st$ — команда вычитает значение в вершине стека $st(0)$ из значения в регистре стека сопроцессора $st(i)$. Результат вычитания запоминается в регистре стека сопроцессора $st(i)$;

fsubp $st(i), st$ — команда вычитает вещественные операнды аналогично команде **fsub** $st(i), st$. Последним действием команды является выталкивание значения из вершины стека сопроцессора $st(0)$. Результат вычитания остается в регистре $st(i-1)$.

Для удобства группы команд вычитания вещественных чисел дополняется командами реверсивного вычитания:

fsubr $st(i), st$ — команда вычитает значение в вершине стека $st(0)$ из значения в регистре стека сопроцессора $st(i)$. Результат вычитания запоминается в вершине стека сопроцессора — регистре $st(0)$;

fsubrp $st(i), st$ — команда производит вычитание подобно команде **fsubr** $st(i), st$. Последним действием команды является выталкивание значения из вершины стека сопроцессора $st(0)$. Результат вычитания остается в регистре $st(i-1)$.

Рассматривая команды умножения вещественных операндов, заметьте, что операнды располагаются исключительно в стеке сопроцессора:

fmul — команда не имеет operandов. Умножает значения в $st(0)$ на содержимое $st(1)$. Результат умножения запоминается в регистре стека сопроцессора $st(0)$;

fmul $st(i)$ — команда умножает значение в $st(0)$ на содержимое регистра стека $st(i)$. Результат умножения запоминается в регистре стека сопроцессора $st(0)$;

fmul $st(i), st$ — команда умножает значения в $st(0)$ на содержимое произвольного регистра стека $st(i)$. Результат умножения запоминается в регистре стека сопроцессора $st(i)$;

fmulp $st(i), st$ — команда производит умножение подобно команде **fmul** $st(i), st$. Последним действием команды является выталкивание значения из вершины стека сопроцессора $st(0)$. Результат умножения остается в регистре $st(i-1)$.

И, наконец, рассмотрим команды, реализующие деление вещественных данных. Подобно командам умножения, операнды этих команд располагаются в стеке сопроцессора:

fdiv — команда (без operandов) делит содержимого регистра $st(0)$ на значение регистра сопроцессора $st(1)$. Результат деления запоминается в регистре стека сопроцессора $st(0)$;

fdiv $st(i)$ — команда делит содержимое регистра $st(0)$ на содержимое регистра сопроцессора $st(i)$. Результат деления запоминается в регистре стека сопроцессора $st(0)$;

fdiv $st(i), st$ — команда производит деление аналогично команде **fdiv** $st(i)$, но результат деления запоминается в регистре стека сопроцессора $st(i)$;

fdivp $st(i), st$ — команда производит деление аналогично команде **fdiv** $st(i), st$. Последним действием команды является выталкивание значения из вершины стека сопроцессора $st(0)$. Результат деления остается в регистре $st(i-1)$.

Для реализации действия деления в сопроцессоре также предусмотрены две реверсивные команды, отличительным признаком которых является наличие символа **r** в качестве последнего или предпоследнего символа мнемокода.

fdivr st(i), st – команда делит содержимое регистра **st(i)** на содержимое вершины регистра сопроцессора **st(0)**. Результат деления запоминается в регистре стека сопроцессора **st(0)**;

fdivrp st(i), st – команда делит содержимое регистра **st(i)** на содержимое вершины регистра сопроцессора **st(0)**. Результат деления запоминается в регистре стека сопроцессора **st(i)**, после чего производится выталкивание содержимого **st(0)** из стека. Результат деления остается в регистре **st(i-1)**.

Разработаем программу вычисления факториала числа 10:

$$Y = \sum_{i=1}^{10} \frac{1}{i}$$

Напомним, что вычисление факториала заключается в выполнении последовательного умножения $1*2*3...*(i-1)*i$.

Листинг 19.4. Исследование вещественных арифметических команд

```
.586p
masm
model    use16 small
.stack   100h
.data    :сегмент данных
;исходный массив
i equ 10
y dq 0
.code
main proc
    mov ax, @data
    mov ds, ax
    finit ;приведение сопроцессора в начальное состояние
    fld1 ;st(0)=1!
    fld1 ;st(0)=i=1, st(1)=1!
    fst y
    mov cx, i-1      ;первый элемент уже вычислили
cycl:
    fld1
    faddp
    fmul st(1), st(0) ;st(0)=i=2, 3..., st(1)=i!
    fld1
    fdiv st(0), st(2) ;1/i!
    fadd y           ;накопление суммы
    fstp y          ;сохранение суммы
    loop cycl
exit:
    mov ax, 4c00h
    int 21h
main endp
end main
```

Дополнительные арифметические команды

В данную группу входят следующие команды:

fsqrt — вычисление квадратного корня из значения, находящегося в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат вычисления помещается в регистр **st(0)**. Следует отметить, что данная команда обладает определенными достоинствами. Во-первых, результат извлечения достаточно точен, во-вторых, скорость исполнения чуть больше скорости выполнения команды деления вещественных чисел **fdiv**;

fabs — вычисление модуля значения, находящегося в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат вычисления помещается в регистр **st(0)**;

fchs — изменение знака значения, находящегося в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат вычисления помещается обратно в регистр **st(0)**. Отличие команды **fchs** от команды **fabs** в том, что команда **fchs** только инвертирует знаковый разряд значения в регистре **st(0)**, не меняя значения остальных бит. Команда вычисления модуля **fabs** при наличии отрицательного значения в регистре **st(0)**, наряду с инвертированием знакового разряда, выполняет изменение остальных бит значения таким образом, чтобы в **st(0)** получилось соответствующее положительное число.

На следующую команду следует обратить особое внимание, так как она позволяет выделить и затем по отдельности проанализировать порядок и мантиссу числа, содержащегося в вершине стека.

fxtract — команда выделения порядка и мантиссы значения, находящегося в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат выделения помещается в два регистра стека — мантисса в **st(0)**, а порядок в **st(1)**. При этом мантисса представляется вещественным числом с тем же знаком, что и у исходного числа, и порядком равным нулю. Порядок, помещенный в **st(1)**, представляется как истинный порядок, то есть без константы смещения, в виде вещественного числа со знаком и соответствует величине *p* формулы (19.1).

Для того чтобы разобраться с тонкостями работы этой команды, изучите листинг 19.5. Исходное десятичное число для программы листинга 19.5 взято из примера 19.3.

Листинг 19.5. Исследование работы команды fxtract

```
.586p
masm
model use16 small
.stack 100h
.data ;сегмент данных
dt dt 0
ch_dt dt 0
y_real dt 45.56 ;y_real=4004 b63d 70a3 d70a 3d71
.code
main proc ;начало процедуры main
    mov ax, @data
```

```

mov    ds, ax
fld    y_real ;st(0)=4004 b63d 70a3 d70a 3d71 (45.56)
fxtract      ;st(1)=4001 a000 ... 0000 (5),
;st(0)=3fff b63d 70a3 d70a 3d71 (1.42375)
fstp   m_dt ;st(0)=4001 a000 ... 0000
;m_dt=3fff b63d 70a3 d70a 3d71 (1.42375)
fstp   ch_dt ;m_dt=3fff b63d 70a3 d70a 3d71 (1.42375)
;ch_dt=4001 a000 ... 0000 (5)

exit:
    mov    ax, 4c00h
    int    21h
main  endp
end main

```

Ведите программу листинга 19.5, получите исполняемый модуль и загрузите его в отладчик Turbo Debugger. Раскройте на весь экран окно Numeric processor, чтобы наблюдать за содержимым регистров стека сопроцессора в шестнадцатеричном формате. В пошаговом режиме проследите за теми действиями, которые выполняет команда `fxtract`. Видно, что команда `fxtract` выделяет характеристику вещественного числа в расширенном формате, по формуле (19.3) вычисляет истинный порядок числа, представляет его как число со знаком и заносит в регистр `st(1)`. Не забывайте, что это порядок нормализованного числа в двоичном формате, поэтому для нашего примера он равен 5 (в формуле 19.2 – $N=2$). Далее команда `fxtract` формирует нулевой порядок значения в `st(0)`, занося в поле характеристики значение `3fffh` (16383_{10}). На этом ее работа заканчивается. Следующие за `fxtract` команды сохраняют значения мантиссы и порядка в памяти как самостоятельные значения. Восстановить обратно исходное значение возможно командой `fscale`, которая к тому же позволяет легко производить умножение на целочисленную степень числа 2.

`fscale` – команда масштабирования – изменяет порядок значения, находящегося в вершине стека сопроцессора – регистре `st(0)` на величину в `st(1)`. Команда не имеет операндов. Величина в `st(1)` рассматривается как число со знаком. Его прибавление к полю порядка вещественного числа в `st(0)` означает его умножение на величину $2^{st(1)}$.

С помощью данной команды удобно масштабировать на степень двойки некоторую последовательность чисел в памяти. Для этого достаточно последовательно загружать числа последовательности в вершину стека, после чего применять команду `fscale` и сохранять значения обратно в памяти. Команда `fscale` является обратной по алгоритму работы команде `fxtract`. Чтобы убедиться в этом, модифицируем листинг 19.5 в программу, представленную в листинге 19.6.

Листинг 19.6. Исследование работы команды `fscale`

```

.586p
masm
model  use16 small
.stack 100h
.data   ;сегмент данных
y_real  dt    45.56 ;y_real=4004 b63d 70a3 d70a 3d71
.code

```

```

main    proc ;начало процедуры main
        mov    ax, @data
        mov    ds, ax
        fld   y_real ;st(0)=4004 b63d 70a3 d70a 3d71 (45.56)
        fxtract      ;st(1)=4001 a000 ... 0000 (5)
        ;st(0)=3fff b63d 70a3 d70a 3d71 (1.42375)
        fscale      ;st(0)= 4004 b63d 70a3 d70a 3d71 (45.56)
exit:
...

```

Сопроцессор имеет программно-аппаратные средства для выполнения операции округления тех результатов работы команд, которые не могут быть точно представлены. Но операция округления может быть проведена и принудительно к значению в регистре `st(0)`, для этого предназначена последняя команда в группе дополнительных команд — команда округления:

`frndint` — команда округления до целого значения — округляет значение, находящееся в вершине стека сопроцессора — регистре `st(0)`. Команда не имеет operandов.

Возможны четыре режима округления величины в `st(0)`, которые определяются значениями в двухбитовом поле `r` с управляющего регистра сопроцессора. Как изменить режим округления? Это выполняется с помощью двух команд `fstcw` и `fldcw`, которые, соответственно, записывают в память содержимое управляющего регистра сопроцессора и восстанавливают его обратно. Таким образом, пока содержимое этого регистра находится в памяти, вы имеете возможность установить необходимое значение поля `r`.

Исследуем работу команды `frndint` на следующем примере (листинг 19.7).

Листинг 19.7. Исследование работы команды `frndint`

```

.586p
masm
model  use16 small
.stack 100h
.data   ;сегмент данных
mem16  dw    0
y_real dt    10.0
x_3dd  3.0
.code
main    proc ;начало процедуры main
        mov    ax, @data
        mov    ds, ax
        fld   y_real
        fld   x_3
        fdiv      ;st(0)=3.333...33
        fstcw mem16
        and   mem16, 111100111111111b
        fldcw mem16
        frndint      ;rc=0, st(0)=3
        fld   y_real
        fld   x_3
        fdiv      ;st(0)=3.333...33
        fstcw mem16

```

```

and    mem16, 1111001111111111b
or     mem16, 1111011111111111b
fldcw mem16
frndint          ;rc=01, st(0)=3
fld    y_real
fld    x_3
fdiv           ;st(0)=3.333...
fstcw mem16
and    mem16, 1111001111111111b
or     mem16, 1111011111111111b
fldcw mem16
frndint          ;rc=10, st(0)=4
fld    y_real
fld    x_3
fdiv           ;st(0)=3.333...
fstcw mem16
and    mem16, 1111001111111111b
or     mem16, 1111111111111111b
fldcw mem16
frndint          ;rc=11, st(0)=3
exit:
    mov    ax, 4c00h
    int    21h
main    endp
end main

```

Рассмотрим еще один пример. Разработаем программу вычисления выражения $z=(\sqrt{|x|}-y)^2$. Результат сохраним в ячейке памяти y в формате двойного слова.

Листинг 19.8. Вычисление выражения $z=(\sqrt{|x|}-y)^2$

```

.586p
masm
model  use16 small
.stack 100h
.data   ;сегмент данных
;исходные данные:
x dd -29e-4
y dq 4.6
z dd 0
.code
main  proc
    mov    ax,@data
    mov    ds,ax
    finit ;приведение сопроцессора в начальное состояние
    fld    x    ;st(0)=x
    fabs  ;st(0)=|x|
    fsqrt
    fsub  y    ;st(0)=sqrt|x|-y
;корень придется вычислять через умножение
    fst    st(1)
    fmul
    fst    z
exit:

```

```
    mov    ax,4c00h
    int    21h
main    endp
end main
```

В процессе написания программы листинга 19.8 выяснилось, что сопроцессор не имеет команды для выполнения операции возвведения в степень. Ниже мы ликвидируем этот пробел в примере программы листинга 19.11.

Команды трансцендентных функций

Сопроцессор имеет ряд команд, предназначенных для вычисления значений тригонометрических функций, таких как синус, косинус, тангенс, арктангенс, а также значений логарифмических и показательных функций. Наличие этих команд значительно облегчает жизнь программисту, вынужденному интенсивно заниматься разработкой вычислительных алгоритмов. Выигрыш налицо. Во-первых, отпадает необходимость самому разрабатывать соответствующие подпрограммы. Во-вторых, результаты трансцендентных команд имеют очень высокую точность. Необходимо обратить внимание читателя на то, что значения аргументов в командах, вычисляющих результат тригонометрических функций, должны задаваться в радианах. В связи с этим приведем правило пересчета. Для нахождения радианной меры угла по его градусной мере необходимо число градусов умножить на $\pi/180$ ($\approx 0,017453$), число минут на $\pi/(180 \times 60)$ ($\approx 0,000291$), а число секунд на $\pi/(180 \times 60 \times 60)$ ($\approx 0,000005$) и найденные произведения сложить.

Ниже перечислены команды трансцендентных функций:

fcos — команда вычисляет косинус угла, находящийся в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат возвращается в регистр **st(0)**.

fsin — команда вычисляет синус угла, находящийся в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат возвращается в регистр **st(0)**.

fsincos — команда вычисляет синус и косинус угла, находящиеся в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат возвращается в регистры **st(0)** и **st(1)**. При этом синус помещается в **st(0)**, а косинус в **st(1)**.

fptan — команда вычисляет *частичный* тангенс угла, находящийся в вершине стека сопроцессора — регистре **st(0)**. Команда не имеет operandов. Результат возвращается в регистры **st(0)** и **st(1)**.

Интересна история этой команды. В отличие от команд вычисления синуса и косинуса, появившихся только в системе команд сопроцессора i387, команда **fptan** присутствовала еще в системе команд сопроцессора i8087. Выполнение ее имело следующую особенность: результат команды возвращался в виде двух значений — в регистрах **st(0)** и **st(1)**. Ни одно из этих значений не является истинным значением тангенса. Истинное его значение получается лишь после операции деления **st(0)/st(1)**. Таким образом, для получения тангенса требуется еще команда деления. Зачем это нужно? Выше мы упомянули о том, что команды для вычисления синуса и косинуса появились только в сопроцессоре i387, поэтому возникает вопрос о том, как вычислялись значения этих функций в ранних сопроцессорах.

Чтобы понять это, введем необходимые обозначения: команда `fptan` вычисляет частичный тангенс числа z , значение которого находится в границах $0 \leq z \leq \pi/4$. Результат работы `fptan`, как уже было отмечено, размещается в двух местах: $x=st(0)$ и $y=st(1)$. Значения x , y и z таковы, что удовлетворяют соотношению $\operatorname{tg}(z/2)=y/x$. Более того, используя значения x и y можно получить значения остальных тригонометрических функций. Для этого используются следующие соотношения:

- $\operatorname{ctg}(z/2)=x/y;$
- $\sin(z)=(2y/x)/((1+(y/x)^2);$
- $\cos(z)=(1-y/x^2)/(1+(y/x)^2).$

Теперь вам, наверное, понятен термин «частичный» тангенс. К счастью, в микропроцессоре i387 появились самостоятельные команды для вычисления синуса и косинуса, вследствие чего отпала необходимость составлять соответствующие подпрограммы. Что же до команды `fptan`, то она по-прежнему выдает два значения в `st(0)` и `st(1)`. Но теперь уже значение в `st(1)` всегда равно единице, это означает, что в `st(0)` находится истинное значение тангенса числа, находившегося в `st(0)` до выдачи команды `fptan`.

`fratan` — команда вычисляет *частичный* арктангенс угла, находящийся в вершине стека сопроцессора — регистре `st(0)`. Команда не имеет операндов. Результат возвращается в регистрах `st(0)` и `st(1)`. Если использовать введенные нами при рассмотрении команды `fptan` обозначения, то для команды `fratan` действует следующее отношение: $z=\operatorname{arctg}(x/y)$.

Значения x и y размещаются в стеке следующим образом: x в `st(0)`, y в `st(1)`. Результат z возвращается в `st(0)`, причем перед этим выполняется выталкивание x и y из стека сопроцессора. Команда `fratan` широко применяется для вычисления значений обратных тригонометрических функций таких, как: `arcsin`, `arccos`, `arcctg`, `arccosec`, `arcsec`. Например, для вычисления функции `arcsin` используется следующая формула:

$$\operatorname{arcsin}(a)=\operatorname{arctg}(a/\sqrt{1-a^2}).$$

Для вычисления этой формулы необходимо выполнить следующую последовательность шагов:

1. Если a является мерой угла в градусах, то выполнить ее преобразование в радианную меру по правилу, приведенному выше.
2. Поместить в стек a в радианной мере.
3. Вычислить значение выражения $\sqrt{1-a^2}$ и поместить его в стек.
4. Выполнить команду `fratan` с аргументами в `st(0)=sqrt(1-a^2)` и `st(1)=a`.

В результате этих действий в регистре `st(0)` будет сформировано значение, которое и будет являться значением `arcsin(a)`.

Аналогично вычисляются значения других обратных тригонометрических функций.

Для вычисления `arccos(a)` используется формула:

$$\operatorname{arccos}(a)=2*\operatorname{arctg}(\sqrt{1-a}/\sqrt{1+a}).$$

Для ее вычисления необходимо выполнить следующую последовательность шагов:

- Если a является мерой угла в градусах, то выполнить ее преобразование в радианную меру по правилу, приведенному выше.
- Вычислить значение выражения $\sqrt{1-a}$ и поместить его в стек.
- Вычислить значение выражения $\sqrt{1+a}$ и поместить его в стек.
- Выполнить команду `fpatan` с аргументами в $st(0)=\sqrt{1+a}$ и $st(1)=\sqrt{1-a}$. В результате этих действий в регистре $st(0)$ сформируется значение, которое и будет являться значением $\arccos(a)$.

Для вычисления $\text{arcctg}(a)$ используется формула:

$$\text{arcctg}(a) = \text{arctg}(1/a) = \text{arctg}(st(1)/st(0)).$$

Для ее вычисления необходимо выполнить следующую последовательность шагов:

- Если a является мерой угла в градусах, то выполнить ее преобразование в радианную меру по правилу, приведенному выше.
- Командой `f1d1` поместить в стек значение 1.
- Поместить в стек значение a .
- Выполнить команду `fpatan` с аргументами в $st(0)=a$ и $st(1)=1$.

В результате этих действий в регистре $st(0)$ сформируется значение, которое и будет являться значением $\text{arcctg}(a)$.

Если вам понадобится вычислить значения других тригонометрических функций, то ход ваших рассуждений должен быть аналогичен приведенному выше. Зная набор функций, поддерживаемых сопроцессором, вам следует искать такие формулы приведения (или выводить их самим), которые выражают нереализованные в сопроцессоре функции через реализованные. Не забывайте контролировать диапазоны значений аргументов для тех команд, которые требуют этого.

Для демонстрации применения команд данной группы реализуем два алгоритма построения графических изображений на основе определенных математических зависимостей. Реализацией этих примеров мы завершим разработку Windows-приложения урока 18. Если вы помните, это было приложение для Windows, в котором остались нереализованными два пункта меню: Графика ▶ Эффекты ▶ Павлин и Графика ▶ Эффекты ▶ Кружева. Рассмотрим вначале суть алгоритмов, в соответствии с которыми строятся эти графические изображения¹.

Фигура «Павлин» состоит из множества отрезков, которые располагаются относительно друг друга в следующем порядке. Один конец всех этих отрезков располагается вдоль горизонтальной линии. Расположение второго конца всех отрезков рассчитывается по определенным формулам, содержащим тригонометрические функции \sin и \cos . Последовательность шагов построения фигуры «Павлин» выглядит следующим образом:

- организуется цикл по переменной $x1$ (координата первого конца каждого отрезка), которая изменяется от 0 до значения $i*cy1$;
- для текущего значения $x1$ вычисляется значения координат другого конца отрезка ($x2, y2$) по формулам.

¹ Математические зависимости для построения этих фигур взяты из книги «Персональный компьютер в играх и задачах». — М.; Наука, 1988.

- $x2=i120+i100*\sin(x1/i30);$
- $y2=i90+i100*\cos(x1/i30);$
- строится отрезок, координаты концов которого располагаются в точках $(x1, icenter)$ и $(x2, y2);$

○ если $x1$ не превысило значения $icyc1$, то цикл повторяется для $x1$, увеличенного на 1.

В этом алгоритме значения $icenter$, $icyc1$, $i90$, $i100$, $i120$, $i30$ представляют собой константы, изменяя которые вы будете получать на экране другие варианты графического изображения «Павлин». На рис. 19.18 показан вид экрана с результатом работы этого алгоритма.

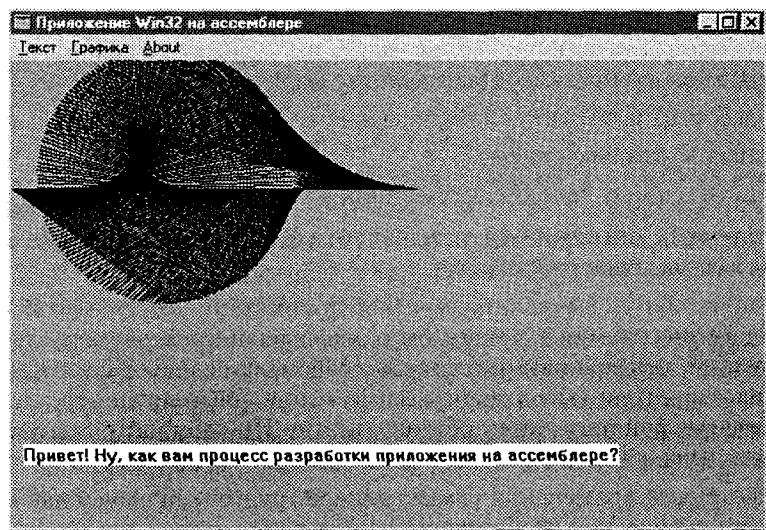


Рис. 19.18. Результат работы приложения (пункт меню Графика ▶ Эффекты ▶ Павлин)

Несущественные детали алгоритма вы легко восстановите по фрагменту программы, приведенному в листинге 19.9. Полный текст программы приведен на диске в каталоге данного занятия.

Листинг 19.9. Фрагмент программы для изображения фигуры «Павлин»

```
.data
;определение констант для фигуры "Павлин"
x1 dd    1
x2 dd    0
y2 dd    0
i30dw   30
i90dw   90
i100    dw    100
i120    dw    120
icenter dd    100
icyc1   dd    319
```

```
...  
.code  
-----MenuProc-----  
;обработка сообщений от меню  
MenuProc proc  
  
@idmreacock: ; "Павлин"  
;очистим окно  
;выполним первичное заполнение раstra серым цветом  
;получим дескриптор серой кисти hbrush=GetStockObject(GRAY_BRUSH)  
    push GRAY_BRUSH  
    call GetStockObject  
    mov @0hbrush, eax  
;выбираем кисть в контекст памяти SelectObject(memdc, @0hbrush)  
    push @0hbrush  
    push memdc  
    call SelectObject  
;заполняем выбранной кистью виртуальное окно  
;BOOL PatBlt(HDC hdc, int nXLeft, int nYLeft, int nWidth, int nHeight, DWORD  
dwRop)  
    push PATCOPY  
    push maxY  
    push maxX  
    push NULL  
    push NULL  
    push memdc  
    call PatBlt  
    mov ecx, icyc1  
    push ecx  
@0m1:  
    finit  
;вычислим x2=120+100*sin(x1/30)  
    pop ecx  
    mov x1, ecx  
    cmp ecx, 0  
    je @0m2  
    dec ecx  
    push ecx  
    fild x1  
    fidiv i30  
    fsin  
    fimul i100  
    fiadd i120  
    fistp x2  
;вычислим y2=120+100*cos(x1/30)  
    fild x1  
    fidiv i30  
    fcov  
    fimul i100  
    fiadd i90  
    fistp y2  
;рисуем отрезок:  
    push NULL
```

```
push icenter
```

Листинг 19.9(продолжение)

```
push x1
push memdc
call MoveToEx
push y2
push x2
push memdc
call LineTo
;генерация сообщения WM_PAINT для вывода строки на экран
push 0
push NULL
push @@hwnd
call InvalidateRect
jmp @@m1
@@m2:
jmp @@exit
```

Фигура «Кружева» формируется следующим образом. В окне приложения строятся вершины правильного многоугольника, количество вершин которого задается константой N . Каждая из этих N вершин соединяется отрезками с другими вершинами. Координаты вершин многоугольника задаются формулами:

$$x_i = x_c + r * \cos\left(\frac{2\pi}{N}i\right)$$

$$y_i = y_c + r * \sin\left(\frac{2\pi}{N}i\right),$$

где i — номер вершины; r — радиус описанной около многоугольника окружности; (x_c, y_c) — координаты центра многоугольника.

В этом алгоритме значения N , r , x_c , y_c представляют собой константы, изменяя которые вы будете получать на экране другие варианты графического изображения «Кружева». На рис. 19.19 показан вид экрана с результатом работы программы.

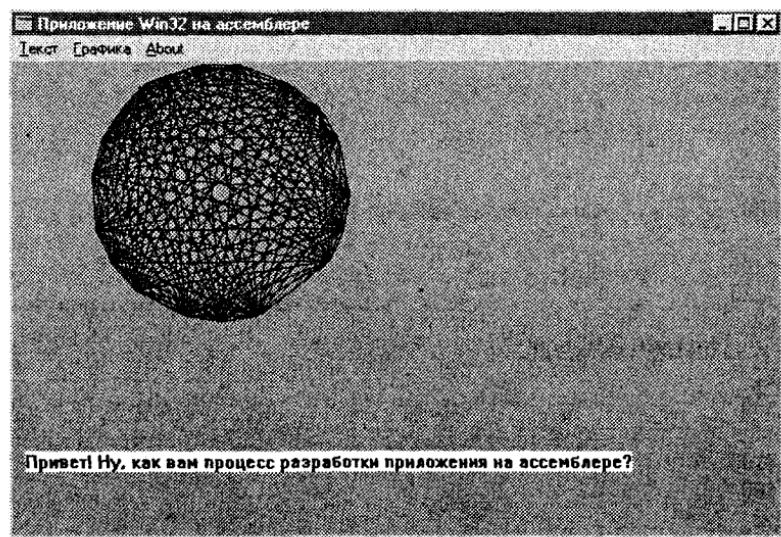


Рис. 19.19. Результат работы приложения (пункт меню Графика ▶ Эффекты ▶ Кружева)

Несущественные детали алгоритма вы легко восстановите по фрагменту программы, приведенному в листинге 19.10. Полный текст программы приведен на диске в каталоге данного занятия.

Листинг 19.10. Фрагмент программы для изображения фигуры «Кружева»

.data

;определение констант для фигуры «Кружева»

;N - число вершин правильного многоугольника

;его можно менять - попробуйте!!!

N equ 18

Xc equ 160

Yc equ 100

masX dd N dup (0)

masY dd N dup (0)

iN dw N

R dw 99

DTTdd 0

t dd 0

i dd 0

j dd 0

i2 dw 2

.code

;-----MenuProc-----

;обработка сообщений от меню

MenuProc proc

@@idmPlaces: ;«Кружева»

;очистим окно

;выполним первичное заполнение раstra серым цветом

;получим дескриптор серой кисти hbrush=GetStockObject(GRAY_BRUSH)

push GRAY_BRUSH

call GetStockObject

mov @@hbrush, eax

;выбираем кисть в контекст памяти SelectObject(memdc, @@hbrush)

push @@hbrush

push memdc

call SelectObject

;заполняем выбранной кистью виртуальное окно

;BOOL PatBlt(HDC hdc, int nXLeft, int nYLeft, int nWidth, int nHeight, DWORD dwRop)

push PATCOPY

push maxY

push maxX

push NULL

push NULL

push memdc

call PatBlt

;вычислим DTT=2π/N

finit

fldpi

```

fidiv iN
fimul i2
fistp DTT
mov t, 0
mov i, 0
;заполняем массивы masX и masY координатами вершин прямоугольника
@0m3:
    mov eax, i
    add eax, DTT
    mov t, eax
    fild t
    fcov
    fistp R
    mov esi, i
    fistp masX[esi*4]
    add masX[esi*4], Xc
    fild t
    fsin
    fistp R
    fistp masY[esi*4]
    mov eax, Yc
    sub eax, masY[esi*4]
    mov masY[esi*4], eax
    inc i
    cmp i, N
    j1 @0m3
;соединим вершины многоугольника:
    mov i, 0
@0m5:
    mov eax, i
    mov j, eax
@0m4:   inc j
;рисуем отрезок:
    push NULL
    mov esi, i
    push masY[esi*4]
    push masX[esi*4]
    push memdc
    call MoveToEx
    mov edi, j
    push masY[edi*4]
    push masX[edi*4]
    push memdc
    call LineTo
    cmp j, N
    j1 @0m4
    inc i
    cmp i, N
    j1 @0m5
;генерация сообщения WM_PAINT для вывода строки на экран
    push 0
    push NULL

```

```
push @@hwnd  
call InvalidateRect  
jmp @@exit
```

Отметим еще две команды сопроцессора:

fprem — команда получения *частичного* остатка от деления. Исходные значения делимого и делителя размещаются в стеке — делимое в **st(0)**, делитель в **st(1)**. Делитель рассматривается как некоторый *модуль*. Поэтому в результате работы команды получается остаток от деления по модулю. Но произойти это может не сразу, так как этот результат, в общем случае достигается за несколько производимых подряд обращений к команде **fprem**. Это происходит, если значения операндов сильно различаются. Физически работа команды заключается в реализации хорошо известного всем действия: деление в столбик. При этом каждое промежуточное деление осуществляется отдельной командой **fprem**. Цикл, центральное место в котором занимает команда **fprem**, завершается, когда очередная полученная разность в **st(0)** становится меньше значения модуля в **st(1)**. Судить об этом можно по состоянию флага **c2** в регистре состояния **swr**:

- если **c2=0**, то работа команды **fprem** полностью завершена, так как разность в **st(0)** меньше значения модуля в **st(1)**;
- если **c2=1**, то необходимо продолжить выполнение команды **fprem**, так как разность в **st(0)** больше значения модуля в **st(1)**.

Таким образом, необходимо анализировать флаг **c2** в теле цикла. Для этого **c2** записывается в регистр флагов основного процессора с последующим анализом его командами условного перехода. Другой способ заключается в сравнении **st(0)** и **st(1)**.

Необходимость в таком частичном исполнении команды **fprem** возникает из-за того, что если делимое слишком велико, а делитель мал, то время получения конечного частичного остатка, удовлетворяющего условию **st(0)<st(1)**, может быть достаточно большим. При этом становится невозможным своевременная реакция на поступающие запросы прерываний, возможно достаточно важные для того, чтобы быть задержанными в обработке.

В контексте рассмотрения нами трансцендентных команд, пример использования **fprem** является наиболее показательным. При рассмотрении команды **fptan** мы упомянули о том, что аргумент **z** должен находиться в диапазоне $0 \leq z \leq \pi/4$. Исходя из того, что данная тригонометрическая функция является периодической, возникает необходимость понижения размерности аргумента **z**, находящегося за рамками указанного выше диапазона допустимых значений для команды **fptan**, что и делается с помощью команды **fprem**.

Важно отметить, что команда **fprem** не соответствует последнему стандарту на вычисления с плавающей точкой IEEE-754. По этой причине в систему команд сопроцессора i387 была введена команда **fpreml**, которая отличается от команды **fprem** тем, что накладывается дополнительное требование на значение остатка в **st(0)**. Это значение не должно превышать половины модуля в **st(1)**. В остальном работа команды **fpreml** аналогична **fprem**.

Команды **fprem** и **fpreml** имеют еще одну особенность, представляющую интерес для команд, вычисляющих значения периодических тригонометрических функ-

ций. После полного завершения работы команды `fprem/fpreml` (когда $c2 \neq 0$), биты $c0, c3, c1$ содержат значения трех младших бит частного, которые логически представляют собой численное значение номера одного из восьми октантов единичного круга. Это, несомненно, важная информация при работе с тригонометрическими функциями.

Далее рассмотрим еще три трансцендентные команды.

`f2xm1` — команда вычисления значения функции $y=2^{x-1}$. Исходное значение x размещается в вершине стека сопроцессора в регистре `st(0)` и должно лежать в диапазоне $-1 \leq x \leq 1$. Результат y замещает x в регистре `st(0)`. Эта команда может быть использована для вычисления различных показательных функций, например, ниже показано, как она используется для возведения числа в степень.

Единица вычитается для того, чтобы получить точный результат, когда x близок к нулю. Вспомните, что нормализованное число всегда содержит в качестве первой значащей цифры единицу. Поэтому, если в результате вычисления функции 2^x получается число 1.00000000456..., то команда `f2xm1`, вычитая 1 из этого числа, и затем, нормализуя результат, формирует больше значащих цифр, то есть делает его более точным. Неявное вычитание единицы командой `f2xm1` компенсируется командой `fadd` с единичным операндом.

`fy12x` — команда вычисления значения функции $z=y\log_2(x)$. Исходное значение x размещается в вершине стека сопроцессора, а исходное значение y — в регистре `st(1)`. Значение x должно лежать в диапазоне $0 \leq x < +\infty$, а значение y — в диапазоне $-\infty \leq y \leq +\infty$. Перед тем как осуществить запись результата z в вершину стека, команда `fy12x` выталкивает значения x и y из стека и только после этого производит запись z в регистр `st(0)`.

Последние две команды сопроцессора очень полезны, в частности, для реализации операции возведения в степень любого числа по любому основанию. Специальной команды в сопроцессоре для операции возведения в степень нет. Поэтому программисту нужно искать подходящую формулу приведения, которая позволит реализовать отсутствующую операцию имеющимися программно-аппаратными средствами сопроцессора. Возведение в произвольную степень числа с любым основанием производится по формуле:

$$x^y = 2^{y * \log_2(x)}$$

Вычисление значения выражения $y * \log_2(x)$ для любого y и $x \geq 0$ производится специальной командой сопроцессора `fy12x`, рассмотренной выше. Вычисление 2^x производится командой `f2xm1` (лишнее действие вычитания 1 пока не замечаем, так как его всегда можно компенсировать сложением с единицей). Но в последнем действии есть тонкий момент, который связан с тем, что величина аргумента x должна лежать в диапазоне: $-1 \leq x \leq 1$. А как быть в случае, если x превышает это значение, (например: 16^3). При вычислении выражения $3 * \log_2(16)$ командой `fy12x` вы получите в стеке значение 12. Попытка вычислить значение 2^{12} командой `f2xm1` ни к чему не приведет — результат будет не определен (по моим наблюдениям значение `st(0)` попросту не изменяется). В этой ситуации логично вспомнить о другой команде сопроцессора `fscale`, которая также вычисляет значение выражения 2^x , но для целых значений x со знаком. Применив формулу $2^{a+b} = 2^a * 2^b$, получаем решение проблемы. Разделяем дробный показатель степени больший $|t|$ на две части — целую и дробную. После этого вычисляем отдельно командами `fscale` и `f2xm1` степени двойки и перемножаем результаты (не забываем компенсировать

вычитание единицы в команде `f2xm1` последующим сложением ее результата с единицей).

Обобщенный алгоритм вычисления степени произвольного числа x по произвольному показателю y следующий (он не претендует на оптимальность решения, так как основное его назначение в раскрытии принципов реализации этой операции):

1. Загрузить в стек основание степени x .
2. Загрузить в стек показатель степени y и проверить его знак:
 - если показатель степени $y < 0$, то запомнить этот факт (далее см. шаг 10) и заменить в стеке значение y на его модуль. Необходимость этого следует из формулы $x^{-y} = 1/x^y$. Перейти на шаг 3.
 - если показатель степени $y \geq 0$, то перейти на шаг 3.
3. Командой `fyl2x` вычислить значение выражения $z = y \lg 2(x)$.
4. Проверить z на диапазон $-1 < z < +1$:
 - если значение $|z| < 1$, то обозначить его как $z2$ и перейти на шаг 5;
 - если значение $|z| > 1$, то представить z в виде двух слагаемых $z = z1 + z2$, где $z1$ — целое значение, а $z2$ — значение меньше единицы. Например, число 16.84 следует представить в виде суммы $16 + 0.84$. В программе это можно выполнить путем циклического вычитания единицы из начального значения z до тех пор, пока оно не станет меньше единицы. Количество вычитаний нужно запомнить, обозначим его n . Перейти на шаг 5.
5. Выполнить команду `f2xm1` с аргументом $z2$.
6. Выполнить команды `f1d1` и `fadd`, компенсирующие единицу, которая была вычтена из результата функции 2^z на шаге 5 предыдущей командой `f2xm1`.
7. Выполнить команду `fscale` с аргументом $z1$. Переменную $z1$ нужно инициализировать нулем, что позволит получить корректный результат, даже если исходное значение z было меньше единицы.
8. Выполнить умножение результатов, полученных на шаге 6 и 7. После этого в вершине стека находится результат возведения в степень.
9. Завершить работу.
10. Шаг выполняется в случае, если показатель степени был отрицательным. В этом случае необходимо единицу разделить на результат шага 8. Для этого в стек загружаем единицу командой `f1d1` и применяем команду `fdiv`. Завершить работу.

Данный алгоритм реализует программа листинга 19.11.

Листинг 19.11. Возведение числа в произвольную степень

```
.586p  
masm  
model use16 small  
.stack 100h  
.data ;сегмент данных  
flag db 0  
p1 dd 0  
y dt 2.0 ;основание степени  
x dt -2.0 ;показатель степени
```

```

.code
main    proc  ;начало процедуры main
        mov   ax, @data
        mov   ds, ax
        finit,
        fld   y
        fld   x
        ftst
        fstsw ax
        sahf
        jnc   m1  ;переход, если x >=0
        inc   flag ;взведем flag, если x<0
        fabs  :|x|
m1: fxch
        fyl2x
        fst   st(1)
        fabs  :|z|
;c сравним |z| с единицей:
        fld1
        fcom
        fstsw ax
        sahf
        jp    exit  ;операнды не сравнимы
        jnc   m2  ;если |z|<1, то переход на m2
        jz    m3  ;если |z|=1, то переход на m3
;если |z|>1, то приводим к формуле z=z1+z2, где z1 - целое, z2 - дробное и z2<1:
        xor   ecx, ecx ;счетчик вычитаний
m12: inc   cx
        fsub  st(1), st(0)
        fcom
        fstsw ax
        sahf
        jp    exit  ;операнды не сравнимы
        jz    m12
        jnc   m2  ;если |z|<1, то переход на m2
        jmp   m12 ;если |z|>1, то переход на m12
m3: mov  p1, 1
        jmp  $+7
m2: mov  p1, ecx
        fxch
        f2xm1
        fadd  ;компенсируем 1
        fldl  p1  ;показатель степени для fscale
        fld1
        fscale
        fxch
        fincstp
        fmul
;проверка на отрицательную степень
        cmp   flag, 1
        jnz   exit
        fld1

```

```
fxch  
fdiv  
exit:  
    mov    ax, 4c00h  
    int    21h  
main    endp  
end main
```

Используя тождество $\log n(x) = \log(n) * \log_2(x)$, можно вычислять логарифм любого числа по любому основанию. Значения $\log_2(2)$, $\log_2(10)$ и некоторых других логарифмических функций вычисляются соответствующими командами сопроцессора: `f1d12t`, `f1d12e`, `f1d1g2`, `f1d1n2`.

`fy12xp1` — команда вычисления значения функции $z = y \log_2(x+1)$. Исходное значение x размещается в вершине стека сопроцессора — регистре `st(0)`, а исходное значение y — в регистре `st(1)`. Значение x должно лежать в диапазоне $0 \leq |x| \leq 1 - 1/\sqrt{2}$, а значение y в диапазоне $-\infty \leq y \leq +\infty$. Перед тем как осуществить запись результата z в вершину стека, команда `fy12xp1` выталкивает значения x и y из стека и только после этого производится запись z в вершину стека — регистр `st(0)`.

Очень интересную программу можно найти на диске, прилагаемой к книге, в подкаталоге ..\OutFPU каталога данного урока. Программа преобразует содержимое вершины стека в символьный вид и выводит его на экран. При этом демонстрируется работа многих уже рассмотренных команд. Рекомендую ознакомиться с этим примером использования команд сопроцессора.

Команды управления сопроцессором

Последняя группа команд предназначена для общего управления работой сопроцессора. Команды этой группы имеют особенность — перед началом своего выполнения они не проверяют наличие незамаскированных исключений. Однако такая проверка может понадобиться, в частности для того, чтобы при параллельной работе основного процессора и сопроцессора предотвратить разрушение информации, необходимой для корректной обработки исключений, возникших в сопроцессоре. Поэтому некоторые команды управления имеют аналоги, выполняющие те же действия плюс одну дополнительную функцию — проверку наличия исключения в сопроцессоре. Эти команды имеют одинаковые мнемокоды (и машинные коды тоже), отличающиеся только вторым символом — символом `n`:

- мнемокод, не содержащий второго символа `n`, обозначает команду, которая перед началом своего выполнения проверяет наличие незамаскированных исключений;
- мнемокод, содержащий второй символ `n`, обозначает команду, которая перед началом своего выполнения не проверяет наличия незамаскированных исключений, то есть выполняется немедленно, что позволяет сэкономить несколько машинных тактов.

Эти команды имеют одинаковый машинный код. Отличие лишь в том, что перед командами, не содержащими символа `n`, транслятор ассемблера вставляет команду `wait`. Команда `wait` является полноценной командой основного процессора и ее, при необходимости, можно указывать явно. Команда `wait` имеет аналог среди

команд сопроцессора — `fwait`. Обеим этим командам соответствует код операции `9bh`:

`wait/fwait` — команда ожидания. Предназначена для синхронизации работы процессора и сопроцессора. Так как основной процессор и сопроцессор работают параллельно, то может создаться ситуация, когда за командой сопроцессора, изменяющей данные в памяти, следует команда основного процессора, которой эти данные требуются. Чтобы синхронизировать работу этих команд, необходимо включить между ними команду `wait/fwait`. Встретив данную команду в потоке команд, основной процессор приостановит свою работу до тех пор, пока не поступит аппаратный сигнал о завершении очередной команды в сопроцессоре. Здесь есть еще один эффект, отмеченный выше. Он связан с корректной обработкой исключений и связанных с ними информацией.

Далее, с необходимой степенью подробности, будут рассмотрены команды, входящие в группу команд управления. Некоторые из этих команд мы уже применяли при разработке программ этого урока.

Из всех команд управления, первой логично рассмотреть команду, приводящую сопроцессор в некоторое начальное состояние:

`finit/fninit` — команда инициализации сопроцессора. Данная команда инициализирует управляющие регистры сопроцессора определенными значениями:

- регистр управления `cwr` инициализируется числом `037h`, что означает установку следующих режимов:
 - поле округления `rc=00` — округление к ближайшему целому;
 - биты 0–5 установлены в единицу, что означает — все исключения замаскированы;
 - поле управления точностью `rc=11` — максимальная точность (64 бита);
- регистр состояния `swr` инициализируется нулевым значением, что означает отсутствие исключений и указание на то, что физический регистр стека сопроцессора `r0` является вершиной стека и соответствует логическому регистру `st(0)`;
- регистр тегов `twr` инициализируется единичным значением — это означает, что все регистры стека сопроцессора пусты;
- регистры указателей данных `dpr` и команд `ipr` инициализируются нулевыми значениями.

Данную команду используют перед первой командой сопроцессора в программе и в других случаях, когда необходимо привести сопроцессор в начальное состояние.

Следующие две команды работают с регистром состояния `swr`.

`fstsw/fnstsw ax` — команда сохранения содержимого регистра состояния `swr` в регистре `ax`. Эту команду целесообразно использовать для подготовки к условным переходам по описанной при рассмотрении команд сравнения схеме;

`fstsw/fnstsw назначение` — команда сохранения содержимого регистра состояния `swr` в ячейке памяти. От рассмотренной выше, команда отличается типом операнда — теперь это ячейка памяти размером 2 байта (в соответствии с размерностью регистра `swr`);

Обратите внимание, что команды, работающие с регистром `swr`, выполняют только считывание содержимого этого регистра. Запись не предусмотрена, так как она

просто не требуется, за исключением, возможно поля `top`, которое является указателем текущей вершины стека. Для изменения этого поля предусмотрены отдельные команды `fincstp`, `fdecstp`, которые будут рассмотрены ниже. Рассмотренные ниже две команды, работающие с информацией в регистре управления `cwr`, напротив, поддерживают действие записи и чтения содержимого этого регистра.

`fstcw/fnscw назначение` — команда сохранения содержимого регистра управления `cwr` в ячейке памяти размером 2 байта. Эту команду целесообразно использовать для анализа полей маскирования исключений, управления точностью и округления. Следует заметить, что в качестве операнда назначения не используется регистр `ax`, в отличие от команды `fstsw/fnshtsw`.

`f1dcw источник` — команда загрузки значения ячейки памяти размером 16 бит в регистр управления `cwr`. Эта команда выполняет действие, противоположное командам `fstcw/fnscw`. Команду целесообразно использовать для задания или изменения режима работы сопроцессора. Следует отметить, что если в регистре состояния `swr` установлен любой бит исключения, то попытка загрузки нового содержимого в регистр управления `cwr` приведет к возбуждению исключения. По этой причине необходимо перед загрузкой регистра `cwr` сбросить все флаги исключений в регистре `swr`.

Следующая команда позволяет сбросить флаги исключений, которые, в частности, необходимы для корректного выполнения команды `f1dcw`. Ее также применяют и в случаях, когда необходимо сбрасывать флаги исключений в регистре `swr`, например, в конце подпрограмм обработки исключений. Если этого не делать, то при исполнении первой же команды сопроцессора прерванной программы (кроме тех команд, которые имеют в названии своего мнемокода второй символ `n`) будет опять возбуждено исключение.

`fclex/fnclex` — команда сброса флагов исключений в регистре состояния `swr` сопроцессора. Команда не имеет operandов.

Как мы уже отметили выше, сопроцессор имеет две команды, которые работают с указателем стека в регистре `swr`.

`fincstp` — команда увеличения указателя стека на единицу (поле `top`) в регистре `swr`. Команда не имеет operandов. Действие команды `fincstp` подобно команде `fst`, но она извлекает значение операнда из стека «в никуда». Таким образом, эту команду можно использовать для выталкивания, ставшего ненужным операнда, из вершины стека. Команда работает только с полем `top` и не изменяет соответствующее данному регистру поле в регистре тегов `twr`, то есть регистр остается занятым и его содержимое из стека не извлекается.

`fdecstp` — команда уменьшения указателя стека (поле `top`) в регистре `swr`. Команда не имеет operandов. Действие команды `fdecstp` подобно команде `f1d`, но она не помещает значения операнда в стек. Таким образом, эту команду можно использовать для проталкивания внутрь стека operandов, ранее включенных в него. Команда работает только с полем `top` и не изменяет соответствующее данному регистру поле в регистре тегов `twr`, то есть регистр остается пустым.

Следующая команда помечает любой регистр стека сопроцессора как пустой:

`ffree st(i)` — команда освобождения регистра стека `st(i)`. Команда записывает в поле регистра тегов, соответствующего регистру `st(i)`, значение `11b`, что соответствует пустому регистру. При этом указатель стека (поле `top`) в регистре

swr и содержимое самого регистра не изменяются. Применяя эту команду, помните, что **st(i)** — это относительный, а не физический номер регистра стека сопроцессора. Необходимость в этой команде может возникнуть при попытке записи в регистр **st(i)**, который помечен, как непустой. В этом случае будет возбуждено исключение. Для предотвращения этого применяется команда **ffree**.

В сопроцессоре есть своя команда пустая операция:

fnop — команда пустая операция. Не производит никаких действий и влияет только на регистр указателя команды **i gr**.

В группе команд управления можно выделить подгруппу команд, работающих с так называемой средой сопроцессора. *Среда сопроцессора* — это совокупность регистров сопроцессора и их значений. Среда сопроцессора может быть *частичной* или *полной* (рис. 19.20). О какой именно среде идет речь, определяется одной из рассмотренных ниже команд.

fsave/fnsave приемник — команда сохранения полного состояния среды сопроцессора в память, адрес которой указан операндом *приемник*. Размер области памяти зависит от размера операнда сегмента кода **use16** или **use32**:

- **use16** — область памяти должна быть 94 байта (рис. 19.20, *a, б*): 80 байт для восьми регистров из стека сопроцессора и 14 байт для остальных регистров сопроцессора с дополнительной информацией;
- **use32** — область памяти должна быть 108 байт (рис. 19.20, *в, г*): 80 байт для восьми регистров из стека сопроцессора и 28 байт для остальных регистров сопроцессора с дополнительной информацией;

Для того чтобы восстановить сохраненную ранее среду сопроцессора применяется следующая команда:

frstor источник — команда восстановления полного состояния среды сопроцессора из области памяти, адрес которой указан операндом *источник*. Сопроцессор будет работать в новой среде сразу после окончания работы команды **frstor**.

Следующие две команды сохраняют частичную среду сопроцессора, в состав которой входят регистры **swr, cwr, twr, dpr** и **i gr**:

fstenv/fnstenv приемник — команда сохранения частичного состояния среды сопроцессора в область памяти, адрес которой указан операндом *приемник*. Размер области памяти зависит от размера операнда сегмента кода **use16** или **use32**. Формат области частичной среды сопроцессора совпадает с форматом области полной среды (см. рис. 19.20), за исключением содержимого стека сопроцессора (80 байт).

f1denv источник — команда восстановления частичного состояния среды сопроцессора содержимым из области памяти, адрес которой указан операндом *источник*. Информация в данной области памяти была ранее сохранена командой **fstenv/fnstenv**.

Команды сохранения среды целесообразно применять в обработчиках исключений, так как только с помощью данных команд можно получить доступ, например, к регистрам **drg** и **i gr**. Не исключено использование этих команд при написании подпрограмм или при переключении контекстов программ в многозадачной среде. Области памяти, содержащие сохраненные среды сопроцессора, есть смысл располагать в стеке основной программы. Тогда, в частности, становится возмож-

ным передать информацию коду, находящемуся в другом кольце защиты основного процессора.

| use16 и R-режим | | | use16 и P-режим | | |
|-------------------------------------|----|------------------------|---------------------|----|--|
| Регистр CWR | 0 | | Регистр CWR | 0 | |
| Регистр SWR | 2 | | Регистр SWR | 2 | |
| Регистр TWR | 4 | | Регистр TWR | 4 | |
| Регистр IPR (0-15) | 6 | | Регистр IPR (0-15) | 6 | |
| IPR (16-19) 0 Код операции (0-10) | 8 | | Селектор CS | 8 | |
| Регистр DPR (0-15) | 10 | | Регистр DPR (0-15) | 10 | |
| DPR (16-19) 0000000000000000 | 12 | | Селектор операнда | 12 | |
| st(0) | 14 | | st(0) | 14 | |
| st(1) | 24 | | st(1) | 24 | |
| ... | | | ... | | |
| st(7) | 84 | | st(7) | 84 | |
| 15 | 11 | 0 | 15 | 0 | |
| <i>а</i> | | | <i>б</i> | | |
| use32 и R-режим | | | | | |
| 0000000000000000 | | | Регистр CWR | 0 | |
| 0000000000000000 | | | Регистр SWR | 4 | |
| 0000000000000000 | | | Регистр TWR | 8 | |
| 0000000000000000 | | | Регистр IPR (0-15) | 12 | |
| 0000 IPR (16-31) | | 0 Код операции(0-10) | | 16 | |
| 0000000000000000 | | | Регистр DPR (0-15) | 20 | |
| 0000 DPR (16-31) | | 0000000000000000 | | 24 | |
| | | st(0) | | 28 | |
| | | st(1) | | 38 | |
| | | ... | | | |
| | | st(7) | | 98 | |
| 31 | 15 | 11 | | 0 | |
| <i>в</i> | | | | | |
| use32 и P-режим | | | | | |
| 0000000000000000 | | | Регистр CWR | 0 | |
| 0000000000000000 | | | Регистр SWR | 4 | |
| 0000000000000000 | | | Регистр TWR | 8 | |
| Регистр IPR (0-31) | | | Регистр IPR (31-47) | 12 | |
| 0000000000000000 | | | Регистр DPR (0-31) | 16 | |
| Регистр DPR (0-31) | | | Регистр DPR (31-47) | 20 | |
| 0000 000000000000 | | | Регистр DPR (31-47) | 24 | |
| | | st(0) | | 28 | |
| | | st(1) | | 38 | |
| | | ... | | | |
| | | st(7) | | 98 | |
| 31 | 15 | 11 | | 0 | |
| <i>г</i> | | | | | |

Рис. 19.20. Структура области памяти с полной средой сопроцессора в различных режимах

Рассмотрение последних команд показало, что сопроцессор может работать не только в реальном, но и в защищенном режиме. Для этого необходимо выполнять его переключение между этими режимами. Эта операция реализуется специальными командами:

`fsetpm` – команда переключения сопроцессора из реального в защищенный режим. Команда не имеет операндов. Действие команды влияет только на выполнение команд сохранения и восстановления среды. Для реального и защищенного режима состав и формат информации среды сопроцессора несколько отличается.

`frstpm` – команда переключения сопроцессора из защищенного в реальный режим. Команда не имеет операндов. Действие команды влияет только на выполнение команд сохранения и восстановления среды.

| use16 и R-режим | | | use16 и P-режим | | |
|--------------------------------------|----|----|-------------------------|----|--|
| Регистр CWR | 0 | | Регистр CWR | 0 | |
| Регистр SWR | 2 | | Регистр SWR | 2 | |
| Регистр TWR | 4 | | Регистр TWR | 4 | |
| Регистр IPR (0–15) | 6 | | Регистр IPR (0–15) | 6 | |
| IPR(16–19) 0 Код операции (0–10) | 8 | | Селектор CS | 8 | |
| Регистр DPR (0–15) | 10 | | Регистр DPR (0–15) | 10 | |
| DPR(16–19) 000000000000 | 12 | | Селектор операнда | 12 | |
| 15 11 0 | | | 15 0 | | |
| use32 и R-режим | | | use32 и P-режим | | |
| 0000000000000000 | | | Регистр CWR | 0 | |
| 0000000000000000 | | | Регистр SWR | 4 | |
| 0000000000000000 | | | Регистр TWR | 8 | |
| 0000000000000000 | | | Регистр IPR (0–15) | 12 | |
| 0000 IPR(16–31) | | | 0 Код операции (0–10) | 16 | |
| 0000000000000000 | | | Регистр DPR (0–15) | 20 | |
| 0000 DPR(16–31) | | | 000000000000 | 24 | |
| 31 | 15 | 11 | 0 | | |
| use32 и P-режим | | | | | |
| 0000000000000000 | | | Регистр CWR | 0 | |
| 0000000000000000 | | | Регистр SWR | 4 | |
| 0000000000000000 | | | Регистр TWR | 8 | |
| Регистр IPR (0–31) | | | Регистр IPR (31–47) | 12 | |
| 0000000000000000 | | | Регистр DPR (0–31) | 16 | |
| Регистр DPR (0–31) | | | Регистр DPR (31–47) | 20 | |
| 0000 000000000000 | | | Регистр DPR (31–47) | 24 | |
| 31 | 15 | 11 | 0 | | |

Рис. 19.21. Структура области памяти с частичной средой сопроцессора в различных режимах

Исключения сопроцессора и их обработка

Важной особенностью сопроцессора является возможность распознавания особых ситуаций, которые могут возникать в процессе вычислений. При этом сопроцессор формирует определенные управляющие сигналы и устанавливает бит в своем регистре состояния `swr`. На уроке 15 мы подробно знакомились с идеологией обработки прерываний в основном процессоре. Согласно приведенной там классификации, исключения являются внутренними прерываниями процессора, возникающими в ходе вычислительного процесса. Особые ситуации, возникающие в сопроцессоре, полностью соответствуют определению понятия исключение.

В сопроцессоре могут возникать шесть типов исключений. Все они были перечислены при обсуждении форматов регистра состояния `swr` и регистра управления

`cwr`. Вспомним основные моменты. Регистр состояния `swr` имеет шесть бит, каждый из которых играет роль флага для определенного типа исключения. При возникновении исключения устанавливается соответствующий флаг в этом регистре (рис. 19.2). Сопроцессор аппаратно позволяет запретить явную обработку любого из этих типов исключений. Для этого регистр управления `cwr` имеет шесть бит, играющих роль масок, установка которых и позволяет запретить возникновение соответствующих исключений. Важно отметить, что запрещение обработки исключения вовсе не означает того, что сопроцессор оставляет вычислительную ситуацию неизменной. При возникновении исключения, которое имеет единичное состояние соответствующего бита в регистре `cwr`, сопроцессор выполняет так называемую *маскированную реакцию*, которая включает в себя некоторую последовательность предопределенных разработчиками микропроцессора действий. Естественно, что они не могли учесть всех потребностей программистов, вплоть до того, что последние, возможно, хотели бы реализовать и нестандартные реакции на исключения. В этом случае программисту необходимо установить соответствующий бит в регистре `cwr`. О том, как нужно реагировать на возникновение незамаскированного исключения, мы поговорим чуть позже. Пока же разберемся с причинами возникновения исключений и маскированными реакциями на них со стороны сопроцессора. Эта информация может быть полезна для того, чтобы разобраться с логическими ошибками при выполнении вашей программы и правильно запрограммировать реакцию на их возникновение.

Исключение недействительная операция

Причиной данного исключения являются ошибки в логике программы, наиболее типичные из которых следующие:

- загрузка операнда в непустой регистр стека сопроцессора;
- попытка извлечения операнда из пустого регистра стека сопроцессора;
- использование операнда с недопустимым для данной операции значением.

При возникновении этого исключения устанавливается флаг `i e` (Invalid Operation) в регистре `swr` (см. рис. 19.2). Это исключение маскируется битом `i m` регистра управления `cwr` (см. рис. 19.4). Исключение недействительная операция возникает при работе со стеком и арифметических вычислениях. В каких именно операциях возникло это исключение, судят по содержимому бита `s f` (Stack Fault – ошибка стека) регистра состояния `swr` (см. рис. 19.2). Если он установлен в единицу, то это говорит о том, что исключение было вызвано ошибкой в работе стека сопроцессора, напротив, нулевое состояние бита `s f` говорит о том, что в команде встретился неверный операнд.

Маскированная реакция на исключение недействительная операция зависит от причины ошибки. Если она возникла в результате некорректной работы стека, то сопроцессор перезаписывает содержимое того регистра стека, обращение к которому вызвало исключение. При перезаписи в него заносится специальное численное значение — спокойное нечисло. Если ошибка возникла в результате недопустимого операнда, то в большинстве ситуаций, в регистре стека сопроцессора возвращается также спокойное нечисло.

Кстати, исключение недействительная операция — это единственное исключение, которое не нужно маскировать. Программисту следует самому вмешиваться в обработку ошибочных ситуаций с помощью вызова соответствующего обработчика.

Деление на ноль

Данное исключение возникает в командах, которые явно или неявно выполняют действие деления. Это такие команды, как `fdiv` и ее варианты — `fyl2x` и `fxtract`. Факт возникновения этого исключения фиксируется флагом `ze` (Zero Divide) регистра состояния сопроцессора `swr` и при необходимости маскируется битом `zm` регистра управления `cwr`. Маскированная реакция сопроцессора заключается в формировании результата в виде знаковой бесконечности.

Денормализация операнда

Исключение данного типа возникает, когда команда пытается выполнить операцию с денормализованным операндом. При этом устанавливается флаг `de` (Denormalized Operand), который маскируется битом `dm` регистра управления `cwr`. Если это исключение замаскировано, то его возникновение приводит только к установке флага `de`, после чего сопроцессор нормализует операнд и вычислительный процесс продолжается дальше. Если исключение денормализованного операнда не замаскировано, то вызывается обработчик исключения, который позволяет произвести необходимую обработку ситуации.

Переполнение и антипереполнение

Ситуации переполнение и антипереполнение возникают в случаях, когда порядок результата слишком велик или слишком мал для формата приемника. При возникновении этих исключений в регистре `swr` устанавливаются флаги `oe` (Overflow) и `ue` (Underflow). Эти исключения маскируются битами регистра управления `cwr` `om` и `im`. Исключения могут возникнуть при работе арифметических команд и команд, преобразующих формат operandов, таких как `fst`.

Маскированная реакция для ситуации переполнения состоит в формировании граничных (максимальных или минимальных) значений, представимых в сопроцессоре или специального численного значения в виде знаковой бесконечности.

Немаскированная реакция на возникновение этих исключений зависит от того, где должен формироваться результат. Если приемник — память, то мантисса результата округляется, а порядок приводится к середине своего диапазона. Если приемник — ячейка памяти, то значение в ней не запоминается, при этом не изменяется и содержимое регистра стека.

Неточный результат

Исключение данного типа возникает в случае, когда результат работы команды нельзя точно представить в формате приемника и его приходится округлять. Например, вычисление любой периодической дроби типа $1/3$, будет приводить к возникновению такого типа исключений. В какую сторону произошло округление, можно судить по значению бита `c1`:

- если `c1=0`, то результат был усечен;
- если `c1=1`, то результат был округлен в большую сторону.

При возникновении этих исключений в регистре `swr` устанавливаются флаг `PE` (Precision). Исключение маскируется битом `rm` регистра управления `cwr`.

Немаскируемая обработка исключений

Как производить такую обработку? Для этого необходимо установить в ноль те флаги в регистре `CWR`, которые соответствуют интересующим нас типам исключений. Далее нужно написать обработчик исключения, который будет реализовывать последовательность действий по корректировке ситуации, приведшей к исключению. Неясным остается вопрос о том, как передать управление обработчику исключения? Для ответа на него нужно разобраться с проблемой взаимодействия между собой процессора и сопроцессора при возникновении исключения.

На уроке 16 мы обсуждали системные регистры микропроцессора. В контексте нашего рассмотрения интерес представляет регистр `CR0`. Он имеет несколько бит, имеющих отношение к сопроцессору (табл. 19.3).

Таблица 19.3. Биты `CR0`, имеющие отношение к сопроцессору

| Положение бита в <code>CR0</code> | Название бита | Назначение |
|-----------------------------------|----------------------------------|--|
| 1 | <code>MP</code> (Math Present) | Сопроцессор присутствует. Должен быть установлен в 1 |
| 2 | <code>EM</code> (Emulation Math) | Эмуляция сопроцессора. Когда <code>EM=1</code> , выполнение любой команды сопроцессора вызывает исключение 7. В программах для микропроцессоров i8086...i386 это позволяет иметь альтернативные участки кода для выполнения одинаковых вычислений с использованием команд сопроцессора (<code>EM=0</code>) и без них (когда в конкретной конфигурации компьютера сопроцессор отсутствует (<code>EM=1</code>)) |
| 3 | <code>TS</code> (Task Switched) | Задача переключена. Предназначен для согласования контекстов основного процессора и сопроцессора. Бит <code>TS</code> устанавливается в 1 при каждом переключении задачи. Состояние этого бита проверяется процессором, если очередной выбранной командой является команда сопроцессора. Если бит <code>TS</code> установлен в 1, то процессор возбуждает исключение 7, обработчик которого выполняет необходимые действия, возможно, по сохранению или восстановлению контекста вычислений с плавающей точкой. С битом <code>TS</code> работает команда <code>CLTS</code> , которая устанавливает значение этого бита в 0 |
| 4 | <code>ET</code> (Extension Type) | Тип расширения. Единичное значение этого бита означает поддержку инструкций сопроцессора |
| 5 | <code>NE</code> (Numeric Error) | Численная ошибка. Бит определяет способ обработки исключений сопроцессора: через сигнал внешнего прерывания или через генерацию исключения (см. далее) |

Начиная с модели i486, процессор и сопроцессор размещаются в одном корпусе. Это упростило организацию связи по управлению между ними. Рассмотрим процессы, протекающие в компьютере при возникновении одного из шести перечис-

ленных выше исключений сопроцессора. При возникновении ситуации исключения сопроцессор устанавливает бит суммарной ошибки `es` в регистре состояния `swr` и формирует на одном из своих выходов сигнал ошибки. Этот сигнал ошибки одновременно воспринимается самим процессором, который генерирует исключение `10h`, и в то же самое время, независимо от процессора, заводится на вход `irq13` программируемого контроллера прерываний, обработчик которого вызывается через вектор прерывания `75h` (урок 15). Таким образом, появление сигнала ошибки на выходе сопроцессора приводит к генерации в основном процессоре двух исключений с номерами `10h` и `75h`.

Исключение `10h` является синхронным, так как вызов его обработчика санкционируется процессором при выполнении команды `wait/fwait`. Данные команды в микропроцессорах `i486` и `Pentium` встроены практически во все команды сопроцессора, за исключением некоторых команд управления, поэтому работа любой команды сопроцессора начинается с выяснения того, было ли зафиксировано какое-нибудь из незамаскированных исключений. Здесь важно то, что контекст вычислительной ситуации после выполнения команды, вызвавшей исключение, полностью сформирован (так как исключение синхронное, то есть ожидаемое) и с ним можно корректно работать.

Если сигнал поступает на вход программируемого контроллера прерываний `irq13`, то обработка исключения `75h` может начаться в процессоре раньше, чем в сопроцессоре закончится работа команды, вызвавшей исключение, то есть в этом случае обработка прерывания асинхронна к вычислительному процессу.

Необходимо отметить влияние бита `pe` на процессы, протекающие в компьютере при возникновении исключения. Его состояние определяет стиль обработки исключения процессором. Если бит `pe=1`, то процессор возбуждает исключение `16` (обработка в стиле `i286` и выше), если `pe=0`, то при возникновении исключения процессор останавливается и ждет прерывания от программируемого контроллера прерываний (обработка в стиле `i8086`). По умолчанию бит `pe` устанавливается в `0`.

На уроке 17 мы рассмотрели распределение прерываний в реальном и защищенном режимах (табл. 17.1). Обратите внимание на номер вектора прерываний (7) — обращение к несуществующему сопроцессору (табл. 19.3). Прерывание появилось в микропроцессоре `i286`, для которого сопроцессор не являлся обязательным устройством. Для того чтобы программа, использующая математические вычисления, была независимой от аппаратной конфигурации конкретного компьютера, составляли два варианта участков кода, на которых эти вычисления выполнялись — с использованием команд сопроцессора и с использованием целочисленных команд. При распознавании в потоке команд инструкций сопроцессора, микропроцессоры `i286` и `i386` проверяли бит эмуляции сопроцессора `em`. Если он был равен 1, то процессор возбуждал исключение 7. Это означало, что сопроцессора в конфигурации компьютера нет, и его функции должны были эмулироваться командами целочисленного устройства. Задача об установке бита `em` ложилась на системное программное обеспечение.

Для микропроцессоров `i486` и `Pentium` состояние вычислительной среды определяется состоянием регистров после выполнения команды `fini` и содержимым регистра `cr0` (биты `pr` и `pe`).

Что должен делать обработчик исключений сопроцессора? Его действия зависят от того, какое незамаскированное исключение им обрабатывается. Следует отметить основные действия, которые необходимы при обработке любого исключения:

- сохранение среды сопроцессора командой `fstenv`. Это необходимо для последующего выяснения причин исключения, а в среде сопроцессора как раз и зафиксировано состояние регистров управления сопроцессором при возникновении исключения;
- сброс установленных битов исключений в регистре `swr`, для предотвращения защелкивания возникновения исключений;
- выяснение типа исключения, приведшего к ситуации исключения. Если незамаскированными являются исключения нескольких типов, то обработчик, путем анализа соответствующих битов в регистре `swr` должен определить их. Заметим, что содержимое `swr` берется из сохраненной при входе в процедуру по команде `fstenv` среды сопроцессора;
- выполнение действий по корректировке ситуации;
- возврат в прерванную программу (командой `iret`).

Но мало выяснить причину исключения — это несложно. Важно, исправив ситуацию, вернуть управление прерванной программе. На уроке 17 мы говорили, что в защищенном режиме работы микропроцессора прерывания и исключения делятся на несколько групп: сбой, ловушка или исключение. Это деление осуществляется в зависимости от того, какая информация сохраняется о месте возникновения прерывания (исключения) и возможности возобновления прерванной программы. Для сопроцессора ситуация аналогична. Здесь информация о месте возникновения исключения зависит от типа исключения. Так, для исключений недействительная операция, деление на ноль и денормализованный операнд запоминается адрес команды, вызвавшей исключение. То есть, ситуация для этих типов исключений распознается и возбуждается до исполнения команды сопроцессора (по классификации для защищенного режима — это сбой). При этом операнды в стеке и в памяти не модифицируются. Для остальных типов исключений ошибочная ситуация распознается после выполнения действий виноватой команды. Это означает, что в стеке в качестве адреса места возникновения исключения запоминается адрес следующей за виноватой команды программы. При этом, операнды в памяти и в регистровом стеке, возможно, будут изменены. Ваши действия должны учитывать эти особенности возникновения различных типов исключений.

Использование отладчика

Отладчик Turbo Debugger предоставляет исчерпывающие возможности для отладки программ, использующих сопроцессор. Для наблюдения за состоянием регистров, составляющих программную модель сопроцессора в среде Turbo Debugger, необходимо открыть специальное окно `Numeric processor`. Для этого выберите пункт главного меню `View ▶ Numeric processor` или нажмите последовательность горячих клавиш `Alt+V ▶ N`. По умолчанию окно появится в минимизированном виде. Для того чтобы раскрыть его, щелкните мышью на стрелке в правом верхнем углу окна `Numeric processor`.

В заголовке окна отображаются четыре сообщения:

1. Модель сопроцессора (автоматически определяется отладчиком).

2. «**IPTR=...**» — сообщение о текущем содержимом указателя команд. Этот указатель содержит физический (20-битный) адрес памяти, по которому расположена последняя выполнявшаяся инструкция сопроцессора.
3. «**OPTR=...**» — сообщение об адресе памяти, к которому обращалась последняя команда сопроцессора (если она имела адресный операнд).
4. «**OPCODE=...**» — сообщение о коде операции, последней исполняемой команды сопроцессора. Интересно отметить то, как формирует отладчик код операции в этом поле. Мы отмечали, что машинный код операции всех команд сопроцессора начинается с одинаковой последовательности битов — 11011, поэтому в поле **OPCODE** эти биты отбрасываются. Например, код команды **f1d** — 0d906h (в двоичном виде — 1101 1001 0000 0110). Убираем пять битов, одинаковых для кода операции каждой команды сопроцессора и получаем то, что видим в поле **OPCODE** заголовка окна Numeric processor — 0106h (0000 0001 0000 0110).

В окне Numeric processor выделяются три области. Сразу заметим, что в отличие от областей окна CPU, области окна Numeric processor нельзя раскрывать отдельно. Основную часть окна Numeric processor занимает область Registers, которая отражает состояние восьми регистров стека сопроцессора **st(0)…st(7)**. Указываются только логические номера регистров. Наиболее полная информация о регистрах стека предоставляется, если окно Numeric processor развернуто. Рассмотрим поля Registers, описывающие состояние каждого из регистров стека сопроцессора. Первое поле показывает состояние регистра, возможные значения в этом поле следующие:

- EMPTY** — «пустой»;
- VALID** — в регистре корректное вещественное число;
- ZERO** — в поле нулевое значение;
- NAN** — в регистре находится специальное численное значение (Not A Number).

Второе поле показывает логический номер регистра стека. Третье поле содержит значения в регистре в виде 80-разрядного числа с плавающей точкой. Четвертое поле показывает содержимое регистра стека в шестнадцатеричном виде.

В ходе отладки вы можете влиять на содержимое регистров стека. Для этого область Registers имеет меню, активизируемое правой кнопкой мыши. В меню три команды:

- ZERO** обнуляет содержимое регистра;
- EMPTY** освобождает регистр стека. Содержимое самого регистра стека не изменяется. Изменению подвергается только тег в регистре тегов, в который заносится значение **11b**;
- CHANGE** вносит некоторое значение в регистр стека. Число должно быть в допустимом формате в соответствии с синтаксисом ассемблера.

Следующую область окна Numeric processor условно можно назвать Control. Область Control содержит совокупность полей, название которых совпадает с названиями битов или полей битов в регистре управления сопроцессором **cwr**. Перечислим эти поля:

- i m** — маска недействительная операция;
- d m** — маска денормализованный операнд;

- **zм** — маска деление на нуль;
- **ом** — маска переполнение;
- **ум** — маска отрицательное переполнение;
- **рм** — маска точность;
- **iem** — маска запрос на прерывание (для i8087);
- **rc** — управление точностью;
- **rc** — управление округлением;
- **ic** — управление значением бесконечность.

Область Registers имеет локальное меню, состоящее всего из одной команды — Toggle. Ее назначение — циклическое изменение содержимого выделенного курсором поля.

Третья область окна Numeric processor — Status — содержит совокупность полей, название которых совпадает с названиями битов или полей битов в регистре состояния сопроцессора **swr**:

- **ie** — ошибка недействительная операция;
- **de** — ошибка денормализованный операнд;
- **ze** — ошибка деление на нуль;
- **oe** — ошибка переполнение;
- **ue** — ошибка отрицательное переполнение;
- **re** — ошибка точности;
- **ir** — маска запрос на прерывание;
- **cc** — код условия (состояние бит c3c2c1c0);
- **st** — указатель вершины стека (поле **top** регистра **swr**).

Область Status имеет локальное меню, состоящее из одной команды — Toggle. Ее назначение — циклическое изменение содержимого выделенного курсором поля.

Сам процесс отладки программы ничем не отличается от процесса отладки для основного процессора.

Общие рекомендации по программированию сопроцессора

В заключение урока сформулируем некоторые общие рекомендации по написанию программ для сопроцессора.

- Первый участок программы, использующий команды сопроцессора, должен начинаться с команды **finit**. Это же относится к случаю, если программа содержит несколько независимых друг от друга участков с командами сопроцессора. Тогда в начале каждого участка должна стоять команда **finit**.
- При написании программы вы должны учитывать то, что процессор и сопроцессор работают параллельно. В этом случае вам необходимо особенно тщательно программировать участки, на которых планируется параллельное вы-

полнение команд обоих процессоров. Особое внимание необходимо обращать на синхронизацию использования общих операндов и обработку возможных исключительных случаев.

- Рекомендуется обработку исключений доверять самому сопроцессору, кроме исключения недействительная операция, что позволит своевременно обнаружить ошибки алгоритма.
- При написании программ следует установить такой режим округления, который позволит получить максимально точный результат.
- Для повышения производительности процессора при передаче данных необходимо использовать директиву even. Ее действие заключается в том, что данные, описываемые следующей за ней одной из директив резервирования и инициализации данных, размещаются по ближайшему адресу, значение которого кратно 2. Так как все типы данных сопроцессора имеют длину, кратную двум, то желательно все ячейки памяти, содержащие значения для обработки сопроцессором, размещать в сегменте данных одним блоком, предваряя их описание директивой even, например:

```
....  
.data  
....  
even  
ch_1 dd 35.78  
ch_2 dt 0987687686  
....
```

- Для описания структуры и организации работы с форматированными объектами удобно использовать такие средства ассемблера для работы с данными, как структуры и записи (урок 12).

Подведем некоторые итоги:

- Математический сопроцессор значительно расширяет возможности компьютера по производству вычислений над числами из очень большого диапазона значений.
- Центром программной модели сопроцессора является регистровый стек, который является наиболее эффективной структурой для программирования вычислительных алгоритмов. Использование стека предполагает, что программист предварительно преобразует исходное выражение в форму ПОЛИЗ. Форма ПОЛИЗ, в частности, используется в трансляторах при разборе и генерации кода различных синтаксических конструкций программы (не только математических выражений).
- Сопроцессор на уровне своей системы команд поддерживает большую номенклатуру типов данных: три формата целых чисел, три формата вещественных чисел, десятичные числа. При разработке вычислительных алгоритмов и подборе для их реализации команд сопроцессора следует помнить, что сопроцессор имеет только один внутренний формат представления данных в виде вещественного числа расширенного формата. По этой причине, команды сопроцессора, работающие с форматами отличными от расширенного, вынужде-

ны выполнять дополнительное преобразование данных. Операция преобразования требует дополнительное количество (и немалое) машинных тактов, что не может не сказаться на общем времени выполнения программы.

- Система команд сопроцессора состоит из нескольких групп, которые удовлетворяют основные потребности программиста при подборе средств для реализации большинства вычислительных алгоритмов. В случае отсутствия поддержки на уровне команд сопроцессора каких-либо математических операций, они достаточно просто могут быть реализованы с помощью математических формул приведения через существующие команды.
- В процессе работы внутри сопроцессора могут возникать различные ситуации, требующие внешнего вмешательства. Их называют исключениями. Исключения разбиты на 6 типов, которым соответствуют по 6 бит в регистрах `swr` и `cwr`. Эти биты предназначены для управления обработкой соответствующих исключений. Биты в `swr` фиксируют возникновение исключений определенного типа. Биты в `cwr` определяют способ обработки возникших исключений. Если при возникновении исключения некоторого типа, соответствующий этому исключению бит в `cwr` равен 1, то это означает, что обработка исключения данного типа замаскирована и сопроцессор должен сам исправить ошибочную ситуацию. Если соответствующий возникшему исключению бит в `cwr` равен 0, то это означает, что программист сам желает исправить ошибочную ситуацию. Для этого он должен написать обработчик исключения.
- Разработку программ удобно вести с использованием отладчика. Turbo Debugger предоставляет полную информацию о состоянии вычислительного процесса, использующего команды процессора и сопроцессора.



УРОК

MMX-технология микропроцессоров Intel

-
- Исторический аспект — два типа MMX-расширения
 - Модель целочисленного MMX-расширения
 - Система команд целочисленного MMX-расширения (Pentium MMX/II/III)
 - Отладка программ целочисленного MMX-расширения
 - Пример применения MMX-технологии
 - Модель потокового MMX-расширения
 - Система команд потокового MMX-расширения (Pentium III)
-

В 1997 году фирма Intel представила свой новый микропроцессор — Pentium® MMX™. По сравнению с предыдущими моделями микропроцессоров этой фирмы — i486 и Pentium®, в его архитектуру были добавлены новые наборы регистров и команд. Pentium® MMX™ стал своеобразной вехой в длинном ряду моделей микропроцессоров семейства i80x86. С его появлением историю развития микропроцессоров этого семейства можно разбить на три этапа:

- 16-разрядные микропроцессоры i8086 и i80286. Эти устройства заложили основу популярности микропроцессоров фирмы Intel. Их система команд содержала около 170 машинных инструкций, включая инструкции сопроцессора;
- 32-разрядные микропроцессоры — i386, i486, Pentium, Pentium Pro. Программная модель этих микропроцессоров практически одинакова и поддерживает порядка 220 команд;
- 32-разрядные микропроцессоры Pentium MMX, Pentium II, Pentium III. Микропроцессоры поддерживают новые технологии обработки данных, которые обеспечиваются введением в их архитектуру дополнительных регистров и команд. Эти новые архитектурные элементы составляют так называемое *MMX-расширение*. Система команд микропроцессоров Pentium MMX и Pentium II насчитывает около 277 команд, а микропроцессора Pentium III — еще на 70 команд больше.

На этом уроке мы обсудим MMX-расширение системы команд микропроцессоров Pentium MMX, Pentium II и Pentium III.

MMX-расширение микропроцессора Pentium предназначено для поддержки приложений, ориентированных на работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции. С данными такого типа обычно работают мультимедийные, графические, коммуникационные программы. По этой причине данное расширение архитектуры микропроцессоров Intel и названо MultiMedia eXtensions (MMX).

С появлением микропроцессора Pentium III следует различать MMX-расширение двух типов — целочисленного устройства и устройства с плавающей точкой. Каждое из них имеет свою программную модель и является независимым от другого. Чтобы различать эти типы расширений, введем следующие условные обозначения: MMX-расширение целочисленного устройства будем называть *MMX-расширением*, а MMX-расширение устройства с плавающей точкой — *XMM-расширением*.

ММХ-расширение архитектуры микропроцессора Pentium

Целочисленное MMX-расширение микропроцессора Pentium представляет собой программно-аппаратное решение, дополняющее архитектуру данного микропроцессора новыми свойствами. Впервые это расширение появилось в микропроцессоре Pentium MMX. В неизменном виде оно поддерживается микропроцессором Pentium. В микропроцессоре Pentium III целочисленное MMX-расширение было дополнено новыми командами. Знакомство с архитектурой целочисленного MMX-расширения удобно вести в рамках модели, основу которой составляют две компоненты — *программная и аппаратная*.

Модель целочисленного MMX-расширения

Основа *программной* компоненты — система команд MMX-расширения (57 команд) и четыре новых типа данных. MMX-команды являются естественным дополнением основной системы команд микропроцессора. Основным принципом их работы является одновременная обработка нескольких единиц однотипных данных одной командой — Single Instruction Multiple Data (SIMD). На рис. 20.1 приведены новые типы данных MMX-расширения.

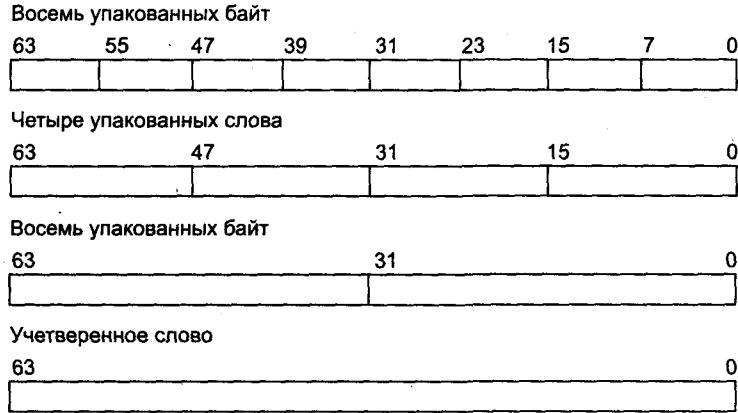


Рис. 20.1. Типы данных, поддерживаемые командами MMX-расширения

На рис. 20.1 видно, что размер поля, занимаемого данными любого из этих типов, одинаков и составляет 64 бита. В них упаковываются и затем используются, как отдельные объекты, данные размером байт, слово и двойное слово (их мы будем называть *элементами* операндов). Именно с такими объектами работают MMX-команды. Данный подход приводит к существенному ускорению обработки данных, прежде всего за счет экономии тактов микропроцессора на передачу данных и выполнение самой команды. Кстати, здесь сказывается преимущество размерности шины данных микропроцессора Pentium, которая равна 64 бита.

Основа *аппаратной* компоненты — восемь новых регистров. Слово «новые» не совсем корректно. На самом деле MMX-расширение использует регистры сопро-

цессора. Как известно, регистры сопроцессора стека имеют размерность 80 бит, что касается регистров MMX-расширения, то их размерность — 64 бита. Поэтому, когда регистры сопроцессора играют роль MMX-регистров, то доступными являются лишь их младшие 64 бита. К тому же, при работе стека сопроцессора в режиме MMX-расширения, он рассматривается не как стек, а как обычный регистровый массив с произвольным доступом. Регистровый стек сопроцессора не может одновременно использоваться и по своему прямому назначению и как MMX-расширение. Забота о его разделении и корректной работе с ним ложится на программиста. Отображение MMX-регистров на регистры стека сопроцессора показано на рис. 20.2.

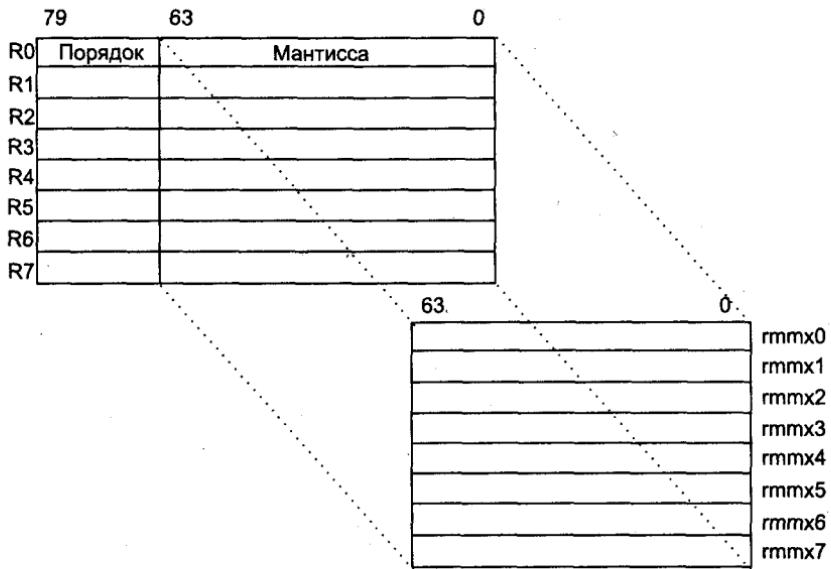


Рис. 20.2. Отображение MMX-регистров на регистры стека сопроцессора

При выполнении MMX-команд сопроцессор переводится в состояние, которое характеризуется следующими признаками:

- регистр тегов сопроцессора обнуляется;
- все регистры стека сопроцессора в MMX-режиме адресуются физически, вне зависимости от значений поля `tos` регистра состояния `swr`. Поле `tos`, кстати, тоже обнуляется;
- MMX-регистру `rmmx0` соответствует физический регистр сопроцессора `r0`, MMX-регистру `rmmx1` — `r1` и т. д. Логическая нумерация регистров сопроцессора не имеет никакого значения;
- содержимое других регистров сопроцессора не изменяется (за исключением применения команды `emms`);
- при записи в MMX-регистр данных, в младшие 64 бита заносятся сами записываемые данные, а в биты 64–79 — единицы. Это делается для того, чтобы при попытке случайного или преднамеренного использования командой сопроцессора MMX-данных, не возникло какого-либо исключения сопроцессора;
- при чтении данных из MMX-регистров их содержимое не изменяется.

Особенности команд MMX-расширение

Важное отличие MMX-команд от обычных команд процессора в том, как они реагируют на ситуации переполнения и заема. На уроке 8 (см. разделы «Сложение двоичных чисел без/со знаком» и «Вычитание двоичных чисел без/со знаком») нами разбирались ситуации, когда результат арифметической операции выходил за размер разрядной сетки исходных operandов. В этом случае производится усечение старших бит результата и возвращаются только те биты, которые умещаются в пределах исходного операнда. Этот принцип формирования результата называется *арифметикой с циклическим переносом* (wgaround arithmetic). Некоторые MMX-команды в подобной ситуации действуют иначе. В случае выхода значения результата за пределы операнда, в нем фиксируется максимальное или минимальное значение. Такой принцип формирования результата называется *арифметикой с насыщением* (Saturation arithmetic). MMX-расширение имеет команды, которые выполняют арифметические операции с использованием обоих принципов. При этом среди них есть команды, учитывающие знаки (значения старших бит) элементов operandов. Принцип формирования результатов арифметических операций с циклическим переносом вы можете посмотреть в уроке 8. Здесь рассмотрим на примерах, как формируются результаты в MMX-командах сложения и вычитания, использующих принцип насыщения.

Пример 20.1. Сложение чисел (беззнаковое насыщение)

$$\begin{array}{r} 254=11111110 \\ + \\ 5=00000101 \\ = \\ 259<>11111111 \end{array}$$

Результат MMX-сложения с беззнаковым насыщением равен 255. При сложении командами микропроцессора `add` и `adc`, использующими принцип циклического переноса, результат равен $00000011=3$, а флаг `c f` устанавливается в 1. Это свидетельствует о факте переполнения.

Пример 20.2. Сложение чисел (знаковое насыщение)

$$\begin{array}{r} +254=11111110 \\ + \\ +5=00000101 \\ = \\ 259<>01111111 \end{array}$$

Результат MMX-сложения со знаковым насыщением двух положительных чисел равен 127. При сложении командами микропроцессора `add` и `adc`, использующими принцип циклического переноса, результат равен $00000011=3$, а флаг `c f` установлен в 1. Это свидетельствует о факте переполнения.

Пример 20.3. Вычитание чисел (беззнаковое насыщение)

$$\begin{array}{r} 05=00000101 \\ - \\ 10=00001010 \\ = \\ -5<> 00000000 \end{array}$$

Результат MMX-вычитания с беззнаковым насыщением двух чисел равен 00h. При вычитании командами микропроцессора sub и sbb, использующими принцип циклического переноса, результат равен 11111011=−5 в дополнительном коде, а флаг с f установлен в 1. Это свидетельствует о факте воображаемого заема единицы из старшего разряда.

Пример 20.4. Вычитание чисел (знаковое насыщение)

+05=00000101

—

+10=000001010

=

−5 <> 10000000

Результат MMX-вычитания со знаковым насыщением двух чисел равен 80h. Это минимально возможное отрицательное число размером в байт. При вычитании командами микропроцессора sub и sbb, использующими принцип циклического переноса, результат равен 11111011=−5 в дополнительном коде, а флаг с f установлен в 1. Это свидетельствует о факте воображаемого заема единицы из старшего разряда.

Подобные рассуждения относятся и к некоторым другим MMX-командам, не являющимся арифметическими. В табл. 20.1 представлены граничные значения насыщения для всех четырех типов данных MMX-расширения.

Таблица 20.1. Граничные значения насыщения MMX-данных

| Типы MMX-данных | Диапазон граничных значений (с насыщением) |
|------------------------------|---|
| Байт без знака | 0...255 (00h...0ffh) |
| Слово без знака | 0...65535 (00h...0ffffh) |
| Двойное слово без знака | 0...4294967295 (00000000...0xffffffffh) |
| Учетверенное слово без знака | (0000000000000000...0xffffffffffffh) |
| Байт со знаком | −128...127 (80h...7fh) |
| Слово со знаком | −32768...32767 (8000h...7fffh) |
| Двойное слово со знаком | −2147483648...2147483647 (80000000...7fffffffh) |
| Учетверенное слово со знаком | (8000000000000000...7fffffffffffffh) |

Отметим некоторые особенности практического использования команд MMX-расширения.

Не все трансляторы языка ассемблера поддерживают MMX-команды. Это относится и к трансляторам TASM версии 5.0 и MASM 6.11. В качестве транслятора, поддерживающего MMX-команды, можно рекомендовать транслятор NASM — продукт фирмы Netwide, информацию о котором можно найти в Интернете по адресу <http://www.cryogen.com/nasm/>. А что же делать тем, кто располагает трансляторами TASM или MASM и не имеет возможности воспользоваться услугами Nasm? Выход здесь один — писать программу в машинных кодах. Ничего себе выход, скажет читатель, после чего закроет книгу, пожалев о потраченных деньгах и времени? Но торопиться с этим не стоит. Дело не такое уж и безнадежное. Фирма Intel предвидела возникновение подобной проблемы и предложила вариант включаемого файла *iammx.inc* для транслятора ассемблера фирмы Microsoft. Он содержит варианты

макроопределений для всех мнемоник MMX-команд. Включив этот файл директивой `include` в начало вашей программы, далее вы можете использовать все команды MMX в определенном формате. С транслятором TASM этот файл использовать нельзя. Для этого его нужно немного доработать. Ситуация усложняется тем, что в пакет ассемблера TASM входят два транслятора: 16-разрядный (`tasm.exe`) и 32-разрядный (`tasm32.exe`). Разрабатываемые с их помощью программы имеют свои особенности. Их достаточно много, поэтому на этом уроке будет изложена методика, используя которую вы сможете разрабатывать программы с использованием MMX-команд для обоих видов трансляторов.

Вполне возможно, что читатель обладает другими средствами для разработки MMX-программ. В этом случае он может использовать материал данного урока в части, касающейся описания форматов MMX-команд и примеров их использования.

Перед началом работы доведем текст включаемого файла `iammx.inc` до рабочего состояния, пригодного для использования с транслятором TASM 5.0. При этом нам придется делать два варианта этого файла – 16- и 32-разрядный. Назовем их, соответственно, `mmx16.inc` и `mmx32.inc`. Они будут использоваться при разработке программ, обрабатываемых 16- и 32-разрядными трансляторами ассемблера (`tasm.exe` и `tasm32.exe`). Эти варианты включаемых файлов предназначены для работы в режиме MASM транслятора TASM 5.0. Объем исходных текстов этих включаемых файлов достаточно велик, поэтому они вынесены на дискету в каталог данного урока. Отличие данных файлов заключается в двух моментах:

- в `mmx16.inc` для моделирования MMX-команд используются 16-разрядные регистры общего назначения, а в `mmx32.inc` используются 32-разрядные регистры;
- в `mmx.inc` для выяснения типа операнда используется оператор `.type`. У автора при попытке использовать эту директиву совместно с `tasm32.exe` возникла ситуация ошибки. Эту ошибку удалось преодолеть только при использовании аналогичного оператора для режима `Ideal - symtype`.

Можно попытаться объединить эти два файла и препроцессорными средствами ассемблера распознавать текущую вычислительную ситуацию, но большого смысла в этом нет, так как 16- и 32-разрядные коды не пересекаются, поэтому проще разработать два файла для каждой из этих технологий разработки и избежать потенциальных ошибок. Так как содержательная часть этих файлов практически одинакова, то есть смысл рассматривать только один из них, отмечая, при необходимости, имеющиеся отличия.

Рассмотрим структуру файла `mmx16.inc`. Этот файл содержит макроопределения, каждое из которых моделирует определенную MMX-команду. В качестве основы для моделирования выступает команда основного процессора. Эта команда должна удовлетворять определенным требованиям. Каковы они? В поисках ответа рассмотрим машинные коды MMX-команд. Видно, что общими у них являются два момента:

1. Поле кода операции MMX-команд состоит из двух байтов, первый из которых равен `0fh`.
2. Все MMX-команды, за исключением команды `emms`, используют форматы адресации с байтами `modR/M` и `sib` и, соответственно, допускают сочетание операндов как обычные двухоперандные команды целочисленного устройства – регистр-регистр или память-регистр.

Для моделирования MMX-команд нужно подобрать такую команду основного процессора, которая удовлетворяет этим двум условиям. Проанализировав с позиций этих требований машинные коды команд основного процессора, выберем для моделирования команды `xadd` и `btr`. В процессе моделирования на место второго байта кода операции этих команд помещается байт со значением кода операции MMX-команды. Когда микропроцессор «видит», что очередная команда является MMX-командой, то он начинает трактовать коды регистров в машинной команде как коды MMX-регистров и ссылки на память, размерностью соответствующей данной команде. В машинном формате команды нет символьических названий регистров, которыми мы пользуемся при написании исходного текста программы, например, `ax` или `bx`. В этом формате они определенным образом кодируются (табл. 20.2). Например, регистр `ax` кодируется в поле `reg` машинной команды как 000 (урок 6). Если заменить код операции команды, в которой одним из операндов является регистр `ax`, на код операции некоторой MMX-команды, то это же значение в поле `reg` микропроцессор будет трактовать как регистр `rmmx0`. Таким образом, в MMX-командах коды регистров воспринимаются соответственно коду операции. В табл. 20.2 приведены коды регистров общего назначения и соответствующих им MMX-регистров. В правом столбце этой таблицы содержится условное обозначение MMX-регистров, принятое в файле `mmx16.inc`. Это же соответствие закреплено рядом следующих определений в этом файле (листинг 20.1).

Листинг 20.1. Фрагмент включаемого файла `mmx16.inc`

```
rmmx0    equ    <ax>
rmmx1    equ    <cx>
rmmx2    equ    <dx>
rmmx3    equ    <bx>
rmmx4    equ    <sp>
rmmx5    equ    <bp>
rmmx6    equ    <si>
rmmx7    equ    <di>
```

В файле `mmx32.inc` эти же определения выглядят, как в листинге 20.2.

Листинг 20.2. Фрагмент включаемого файла `mmx32.inc`

```
rmmx0    equ    <eax>
rmmx1    equ    <ecx>
rmmx2    equ    <edx>
rmmx3    equ    <ebx>
rmmx4    equ    <esp>
rmmx5    equ    <ebp>
rmmx6    equ    <esi>
rmmx7    equ    <edi>
```

Теперь в исходном тексте программы можно использовать символические имена MMX-регистров в качестве аргументов макрокоманд, моделирующих MMX-команды.

Рассмотрим, как в файле `mmx16.inc` описано макроопределение для моделирования MMX-команды упаковки с насыщением слов в байты `packsswb` (листинг 20.3).

Таблица 20.2. Кодировка регистров в машинном коде команды

| Код в поле reg | Регистр целочисленного устройства | MMX-регистр |
|----------------|-----------------------------------|-------------|
| 000 | ax/eax | rmmx0 |
| 001 | cx/ecx | rmmx1 |
| 010 | dx/edx | rmmx2 |
| 011 | bx/ebx | rmmx3 |
| 100 | sp/esp | rmmx4 |
| 101 | bp/ebp | rmmx5 |
| 110 | si/esi | rmmx6 |
| 111 | di/edi | rmmx7 |

Листинг 20.3. Фрагмент включаемого файла mmx16.inc

```

<1> packsswb    macro dest:req,src:req
<2> local        pre,post
<3> pre:
<4> xadd         src,dest
<5> post:
<6> org          pre+1
<7> db            _opc_packsswb
<8> org          post
<9> endm

```

Понимание структуры приведенного макроопределения не должно вызвать у читателя трудностей. Центральное место в макроопределении занимает команда целочисленного устройства (в данном случае `xadd`) и директива `org`. Директива `org` предназначена для изменения значения счетчика адреса. С этой директивой мы уже встречались на уроке 10. В строке 6 директива `org` устанавливает значение счетчика адреса равным адресу `pre+1`. Адрес метки `pre` является адресом первого байта машинного кода команды `xadd`. Соответственно, если этот адрес увеличить на 1, то получим адрес второго байта кода операции. Таким образом, значением текущего счетчика адреса в строке 7 будет адрес `pre+1`. Директива `db` в строке 7 размещает по этому адресу байтовое значение `_opc_packsswb`, которое соответствует второму байту кода операции MMX-команды `packsswb` (листинг 20.1). Директива `org` в строке 8 устанавливает значение счетчика адреса равным значению адреса следующей после `xadd` команды. Для доошных читателей заметим еще один характерный момент. Для его полного понимания необходимо хорошо представлять себе формат машинной команды и назначение ее полей. Достаточно полная информация об этом приведена в материале урока 6 и в Справочнике. Обратите внимание на порядок следования операндов в заголовке макрокоманды, который построен по обычной схеме: `коп назначение, источник`. В команде `xadd` порядок обратный. Этого требует синтаксис команды. Это хорошо поясняет назначение бита `d` во втором байте кода операции, который характеризует направление передачи данных в микропроцессор (то есть в регистр) или в память (из микропроцессора (регистра)). Вы можете провести эксперимент. Проанализируйте машинные коды команды `mov`:

```
.data  
p dw 0  
.code
```

```
...  
mov p,bx ;машиинный код: 89 1e 00 00, d=1, w=1  
mov bx,p ;машиинный код: 8b 1e 00 00, d=0, w=1
```

Из приведенного фрагмента видно, что изменение типов источника и назначения на обратный влияет только на изменение поля кода операции, а именно на его второй бит. Это бит *d* (Directory – направление), который определяет направление передачи. Если *назначение* – регистр, то бит *d*=1 – это означает передачу из памяти в микропроцессор (регистр). И, наоборот, если *назначение* – адрес памяти, то бит *d*=0 – это означает передачу из микропроцессора (регистра) в память. Таким образом, значения полей в остальных байтах машинного кода операции лишь определяют местоположение operandов и не зависят от реального направления передачи. Например, значение второго байта кода операции *opc_packsswb*=63h (0110 0011b). Он имеет значение бита *d*=1, то есть данные передаются из регистра в память. Это нам и позволило в команде *xadd* изменить порядок следования operandов (иначе транслятор ассемблера эту команду не пропустит – подумайте, почему?).

В связи с обсуждаемым моментом интересно посмотреть макроопределение другой MMX-команды – *movd* (листинг 20.4).

Листинг 20.4. Фрагмент включаемого файла mmx16.inc

```
<1> movd      macro dest:req, src:req  
<2> local     pre, post  
<3> if (.type(dest)) and 10h  
<4> pre:  
<5>   xadd      src, dest  
<6> post:  
<7>   org       pre+1  
<8>   db        _opc_movd_ld  
<9>   org       post  
<10> else  
<11> pre:  
<12>   xadd      dest, src  
<13> post:  
<14>   org       pre+1  
<15>   db        _opc_movd_st  
<16>   org       post  
<17> endif  
<18> endm
```

Команда *movd* позволяет производить передачу в обоих направлениях: память – микропроцессор и микропроцессор – память. Для того чтобы правильно смоделировать машинное представление MMX-команды, необходимо определить, каким объектом является operand *dest* (назначение) – ячейкой памяти или регистром. После этого будет ясен тип второго операнда, так как реализованы могут быть лишь две схемы расположения operandов: регистр-регистр и память-регистр. Для выяснения типа операнда ассемблер предоставляет оператор *type*, который имеет синтаксис:

.type выражение

Оператор `.type` в зависимости от типа операнда, указанного в качестве операнда `выражение`, возвращает байтовые значения, которые приведены в табл. 20.3. В строке 3 листинга 20.3 определяется тип операнда `dest` и, если это регистр, то формируется один код операции (строки 4–10), если нет, то — другой (строки 11–17). Кстати, `movd` — это единственная MMX-команда, которая выпадает из общей схемы построения MMX-команд, так как только она позволяет производить обмен с 32-разрядным регистром общего назначения.

Таблица 20.3. Операнды и возвращаемые значения оператора `.type`

| Значения бит | Тип объекта |
|--------------|--|
| 0000 0000b | Указатель адреса памяти в сегменте кода |
| 0000 0010b | Указатель адреса памяти в сегменте данных |
| 0000 0100b | Константа |
| 0000 1000b | Выражение использует режим прямой адресации |
| 0001 0000b | Регистр |
| 0010 0000b | Идентификатор |
| 1000 0000b | Внешний идентификатор |
| ???? ?00?b | Выражение использует режим косвенной адресации |

Адреса operandов в памяти формируются таким же образом, как и для целочисленных команд, а их трактовка зависит от конкретной MMX-команды. Более детально с моделированием остальных MMX-команд вы можете познакомиться, изучив тексты включаемых файлов `mmx16.inc` и `mmx32.inc`, которые находятся на диске, прилагаемой к книге.

Далее мы рассмотрим характеристику MMX-команд, сопровождая описание характерными примерами, демонстрирующими механизм работы команд. В конце данной части урока рассмотрим типовой пример использования MMX-команд.

Система команд

MMX-команды по функциональному признаку делятся на следующие группы (рис. 20.3):

Рассмотрим выделенные классы команд, укажем особенности их применения на примерах.

Команды передачи данных

MMX-команды пересылки, подобно их целочисленным аналогам, являются наиболее часто используемыми. Эти команды осуществляют доставку информации в (из) MMX-регистры (ов). MMX-команды пересылки работают с 32- и 64-разрядными operandами. В данную группу входят следующие команды:

`movd` *приемник*_{источник} — пересылка 32 битов из *источника* в *приемник*. Один из operandов, *источник* или *приемник*, но не одновременно, должен быть MMX-регистром. Другой operand должен быть 32-разрядным регистром или 32-разрядной ячейкой памяти;

movq приемник|источник – пересылка 64 битов из **источника** в **приемник**. В отличие от команды **movd**, оба операнда команды **movq** могут быть MMX-регистрами. Если же операнды смешанные, то один из operandов, **источник** или **приемник**, является MMX-регистром, а другой operand должен быть адресом 64-разрядной ячейки памяти.

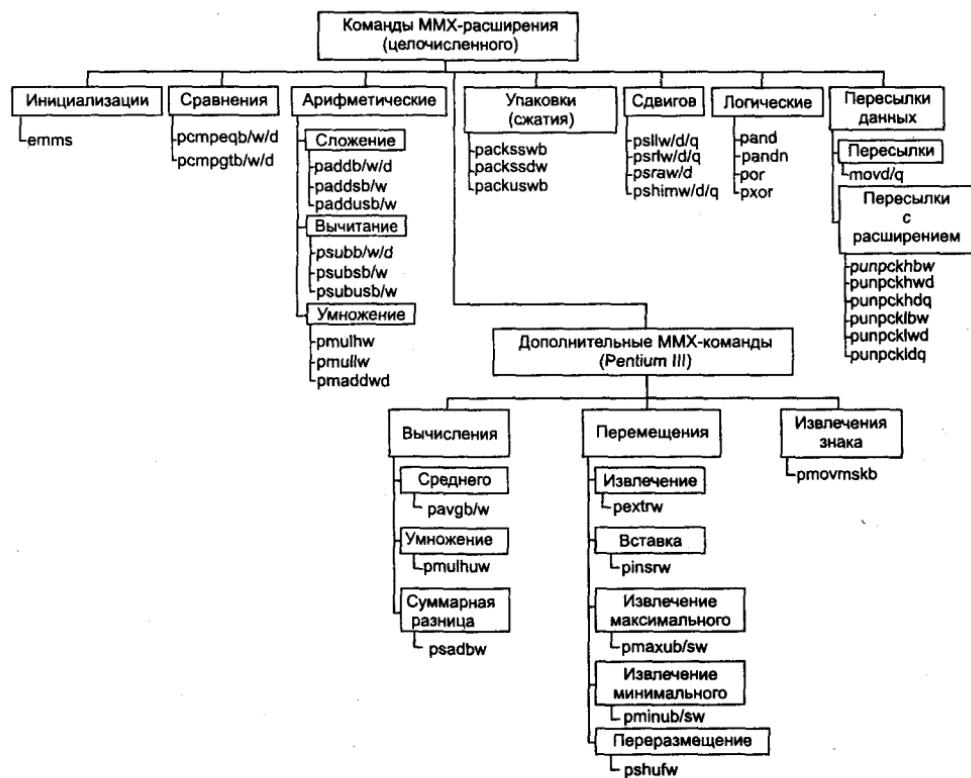


Рис. 20.3. Классификация целочисленных MMX-команд

Команда **movd** работает только с младшей половиной MMX-регистра. Для доступа к старшей половине MMX-регистра необходимо использовать либо сдвиг, либо команду **movq**. Команда **movd** является единственной MMX-командой, допускающей использование в качестве operandов 32-разрядных регистров общего назначения. Это же обстоятельство является причиной того, что при использовании в качестве **приемника** регистра общего назначения макрокоманда **movd** будет работать неправильно. Для того чтобы понять, в чем здесь дело, посмотрим еще раз на макроопределение, моделирующее эту MMX-команду:

```

movd macro dest:req, src:req
  local pre, post
  if (.type(dest)) and 10h
  pre:
    xadd src, dest
  post:
    org pre+1
  
```

```
db    _opc_movid_ld
org  post
else
pre:
  xadd dest, src
post:
  org  pre+1
  db   _opc_movid_st
  org  post
endif
endm
```

Допустимые сочетания операндов для команды `movd` следующие:

```
movd  mem32, rmmx
movd  rmmx, mem32
movd  rmmx, r32
movd  r32, rmmx
```

Приведенное макроопределение хорошо работает при первых трех сочетаниях операндов, а в последнем случае оно перестает понимать, что от него хотят, и дает сбой. Здесь две причины. Во-первых, оператор `.type` (`symtype` в режиме `Ideal`) одинаково реагирует на сочетания `movd rmmx, r32` и `movd r32, rmmx`, так как в результате макроподстановки `rmmx` заменяется на один из регистров `r32`. Поэтому в обоих случаях генерируется команда `xadd rmmx, r32`, со вторым байтом кода операции равным `6eh`. MMX-команда с этим кодом операции предполагает пересылку 32 битов из памяти или регистра общего назначения, но не наоборот. Для выполнения пересылки из регистра общего назначения (32-битной ячейки памяти) в MMX-регистр требуется машинная команда со вторым байтом кода операции равным `7eh`. Вторая причина является следствием первой. Так как MMX-регистры и регистры общего назначения кодируются в машинной команде одинаково, то конкретная машинная команда различает, какой из операндов является MMX-регистром, а какой регистром общего назначения по заложенной в ней схеме и для указанного выше сочетания операндов делает это неправильно. Как выйти из этого положения? Можно, конечно, выполнить дополнительный анализ передаваемых в макрокоманду аргументов, но мы, учитывая единичность этого случая, пошли по самому легкому пути — включили в `mms.inc` дополнительное макроопределение `mov`. Это макроопределение специально предназначено для случая пересылки данных из MMX-регистра в 32-битный регистр общего назначения.

Следующий важный вопрос связан с тем, как описывать данные для работы с MMX-командами. При этом также приходится учитывать то обстоятельство, что при использовании пакета TASM (как, впрочем, и MASM) MMX-команды приходится моделировать. Главное здесь добиться для себя понимания последовательности этого моделирования: на этапе трансляции моделируются операнды целочисленной команды (в нашем случае это `xadd`) и формируется поле кода операции в машинной команде (ее второй байт). Далее на этапе исполнения программы выполняется должная интерпретация операндов команды. Если посмотреть на описание команды `xadd`, то легко увидеть, что она может работать максимум с 32-разрядными операндами. А как же быть, если требуется моделировать MMX-команду, манипулирующую 64-разрядными операндами? В этом случае приходится в очередной раз заниматься обманом. Например, следующий фрагмент программы будет ошибочным:

```
include mmx.inc
.data
mem64 dq 1111222233334444h
...
.code
...
    movq rmmx0, mem64
```

Ошибка возникнет из-за того, что на этапе трансляции будет смоделирована команда `xadd ax, mem64`, которую транслятор не пропустит из-за несовпадения типов операндов. Суть «обмана» состоит в следующем описании данных:

```
include mmx.inc
.data
mem64 dw 4444h
df 111122223333h
...
.code
...
    movq rmmx0, mem64
```

Имейте в виду, что при описании операнда в памяти продолжает действовать принцип размещения данных в памяти «младший байт по младшему адресу». При работе с 32-разрядным ассемблером `tasm32.exe` описание данных должно быть выполнено так:

```
include mmx32.inc
.data
mem64 dd 33334444h
dd 11112222h
...
.code
...
    movq rmmx0, mem64
```

В листинге 20.5 приведены примеры использования команд пересылки данных. Текст программы листинга используйте в качестве основы для разработки других программ урока. Далее для экономии места мы будем приводить только фрагменты соответствующих программ.

Листинг 20.5. Использование MMX-команд пересылки данных

```
<1> ;-----mmx16.asm-----
<2> .586p
<3> model      use16 small     ; use16 обязательно
<4> %NOINCL   ; запретить вывод текста включаемых файлов
<5> include mmx16.inc
<6> .stack      100h
<7> .data       ;сегмент данных
<8> mem        dw 4444h
<9> df         111122223333h
<10> mem1      dw 3h
<11>           df 0002cccc0004h
<12> mem2      dw 0h
```

```

<13>          df      0001dddd0f03h
<14> mem3       dw      7fffh
<15>          df      0001dddd0f03h
<16> .code
<17> main        proc   ;начало процедуры main
<18>           mov    ax, @data
<19>           mov    ds, ax
<20>
<21>           movd   rmmx0, mem  ;rmmx0=0000 0000 3333 4444
<22>           movdr32  ax, rmmx0  ; rmmx0=0000 0000 3333 4444, eax=3333
                                         4444
<23>           movq   rmmx0, mem1 ;rmmx0=0002 cccc 0004 0003
<24>           movd   mem2, rmmx0 ;mem2=0300 0400 cccc 0200, rmmx0=0002
                                         cccc 0004 0003
<25>           pxor   rmmx0, rmmx0 ;rmmx0=0000 0000 0000 0000
<26>           movd   mem3, rmmx0 ;mem3=0000 0000 0000 0000
<27>           emms
<28>           mov    ax, 4c00h ;пересылка 4c00h в регистр ax
<29>           int    21h   ;вызов прерывания с номером 21h
<30> main        endp   ;конец процедуры main
<31> end         main   ;конец программы с точкой входа main

```

Обратите внимание на строку 22, в которой команда `movd` (ранее мы уже обсуждали эту команду и соответствующую ей макрокоманду `movdr32`) пересыпает 32 бита в регистр общего назначения, но при этом указывается 16-разрядный регистр `ax`. Указать впрямую `eax`, нельзя, так как для 16-разрядного ассемблера вместо `rmmx0` в ходе моделирования подставляется `ax` (листинг 20.1). На этапе выполнения все расставляется на свои места, так как микропрограмма, реализующая MMX-команду `movd` по полу кода операции и коду регистра, который, как видно из табл. 20.3, одинаков для регистров `ax` и `eax`, производит требуемую пересылку — из `rmmx0` в `eax`.

Здесь уместно уделить внимание еще одной проблеме, которая неизбежно встает на определенном этапе разработки программы — какие отладочные средства можно использовать для просмотра результатов работы MMX-команд? Можно попробовать подыскать отладчик, поддерживающий работу с MMX-расширением, но это делать совсем необязательно. Понимание того, что MMX-расширение архитектурно создано на базе сопроцессора, дает вам возможность использовать практически любой отладчик для работы с MMX-приложениями, в том числе Turbo Debugger. При этом, конечно, необходимо будет соблюдать определенные ограничения, источник возникновения которых в том, что Turbo Debugger ничего не знает о MMX-командах. Это обстоятельство может приводить к его неадекватной работе. Но если следовать некоторым рекомендациям, то можно пройти над подводными камнями, не задев их.

Отладка программ

В пакете TASM версии 5.0 есть три варианта отладчика Turbo Debugger:

- `td.exe` — отладчик для разработки 16-разрядных программ;

- **td32.exe** — отладчик для разработки 32-разрядных программ;
- **tdw.exe** — отладчик для разработки Windows-приложений.

В основном, все сказанное ниже касается любого из этих трех отладчиков, так как проблема у них общая — непонимание мнемоники MMX-команд.

Чтобы создать исполняемый файл, пригодный для отладки, задайте в качестве параметра транслятора (**tasm.exe** или **tasm32.exe**) ключ **/zi**, а компоновщика (**tlink.exe** или **tlink32.exe**) — ключ **/v**. Загрузите исполняемый файл в отладчик (исходный текст появится в окне **Module**).

Теперь разделите программу на участки двух типов: содержащие MMX-команды и не содержащие их. С участками, не содержащими MMX-команд, вы можете работать в обычном режиме. Для отладки программы на участках, содержащих MMX-команды, перейдите в окно **CPU** (View ▶ CPU), а также откройте окно **Numeric processor** (View ▶ Numeric processor) (рис. 20.4).

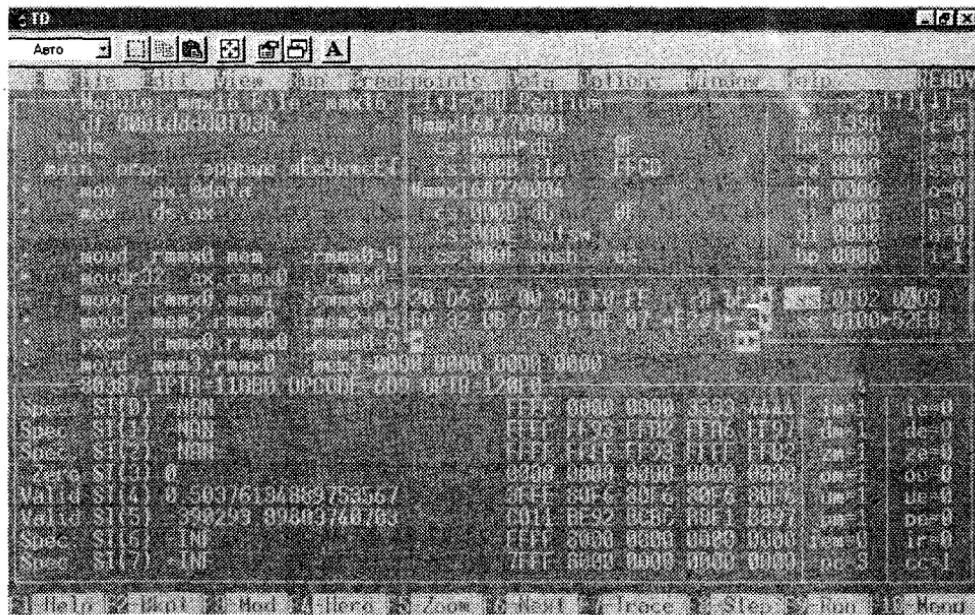


Рис. 20.4. Вид отладчика при работе с MMX-командами

Сам процесс отладки традиционен и вы можете управлять им, используя привычные средства Turbo Debugger.

Арифметические команды

В данную группу входят команды для реализации трех основных арифметических операций: сложения, вычитания, умножения. При этом, как мы уже отметили выше, для операций сложения и вычитания предусмотрены команды, учитывающие знак операндов (значение старшего бита).

Команды сложения

Команды сложения делятся на две подгруппы исходя из того, как формируется результат при возникновении переполнения — по принципу насыщения или циклического переноса.

`paddb | paddw | paddd приемник, источник` — сложение беззнаковых упакованных байт, слов, двойных слов. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу циклического переноса (см. раздел «Особенности MMX-команд»). Перенос теряется и нигде не учитывается.

```
.data ;сегмент данных
mem    dw    4444h
      df    111122223333h
mem1   dw    0ffffeh
      df    0fffccfffdffffh
.code
.....
      movq  rmmx0, mem   ;rmmx0=11 11 22 22 33 33 44 44
                        ;mem1=ff fc ff fd ff ff fe
      paddb rmmx0, mem1 ;rmmx0=10 0d 21 1f 32 32 43 42
....
```

`paddsbt | paddsw приемник, источник` — сложение упакованных байт и слов со знаком. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу знакового насыщения (см. раздел «Особенности MMX-команд»).

```
.data ;сегмент данных
mem    dw    4444h
      df    111122223382h
mem1   dw    0157eh
      df    0717c3f7d7ffah
.code
.....
      movq  rmmx0, mem   ;rmmx0=11 11 22 22 33 82 44 44
                        ;mem1 =71 7c 3f 7d 7f fa 15 7e
      paddsb rmmx0, mem1 ;rmmx0=7f 7f 61 7f 7f 80 59 7f
....
```

Исходные данные этого примера подобраны так, чтобы было видно, как при переполнении формируется результат по принципу знакового насыщения. Сравните их с теми результатами, которые получаются в результате работы следующих команд.

`paddusb | paddusw приемник, источник` — сложение беззнаковых упакованных байт и слов. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу беззнакового насыщения.

```
.data ;сегмент данных
mem    dw    4444h
```

```
        df      111122223382h  
mem1  dw      0157eh  
        df      0717c3f7d7ffah  
.code  
... ... ...  
        movq    rmmx0, mem   ;rmmx0=11 11 22 22 33 82 44 44  
                           ;mem1 =71 7c 3f 7d 7f fa 15 7e  
        paddusb    rmmx0, mem1 ;rmmx0=82 8d 61 9f b2 ff 59 c2
```

Команды вычитания

Набор команд вычитания аналогичен командам сложения и содержит три группы. Эти команды вычитывают беззнаковые элементы традиционным для целочисленного устройства способом — по принципу циклического переноса, знаковые элементы по принципу знакового насыщения и беззнаковые элементы по принципу беззнакового насыщения.

psubb | **psubw** | **psubd** *приемник, источник* — вычитание беззнаковых упакованных байтов, слов, двойных слов. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу циклического переноса, то есть так, как это делается командами **sub** и **sbb** микропроцессора. Заем из старшего разряда, естественно, теряется и нигде не учитывается.

```
.data ;сегмент данных
mem dw 4444h
      df 111122223382h
mem1 dw 157eh
      df 1123f7d7ffah
.code
... ...
    movq rmmx0, mem ;rmmx0=1111 2222 3382 4444
                      ;mem1 =1123 f7d7 ffah 157e
    psubw rmmx0, mem1 ;rmmx0=0fff e2a5 b388 2ec6
```

psubsb | **psubsw** *приемник, источник* – вычитание упакованных байт и слов со знаком. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* – либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении ситуации, когда результат вычитания получается меньше 80h (8000h), поле соответствующего байта (слова) формируется по принципу знакового насыщения (в нем остается значение 80h (8000h)). Если результат больше 7fh (7fffh), то результат насыщается до значения 80h (8000h).

```
.data ;сегмент данных
mem dw 5h
      df 0ffb22223382h
mem1 dw 8003h
      df 7ffedf7d7ffah

.code
      movq rmmx0, mem ;rmmx0=ffb 2222 3382 0005
                      ;:mem1=7ffe df7d 7ffa 8003
```

`psubsw rmmx0, mem1 ;rmmx0=8000 42a5 b388 7fff`

Первая и последняя шестнадцатеричные тетрады показывают результат, сформированный в соответствии с принципом знакового насыщения.

`psubusb | psubusw приемник,источник` – вычитание беззнаковых упакованных байтов и слов. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* – либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении ситуации, когда результат вычитания получается меньше 00h (0000h), поле соответствующего байта или слова формируется по принципу беззнакового насыщения (в нем остается значение 00h (0000h)).

```
.data ;сегмент данных
mem    dw      5h
       df      0ffb22223382h
mem1   dw      8003h
       df      7ffe0f7d0ffah
.code
...
        movq  rmmx0, mem   ;rmmx0=ffffb 2222 3382 0005
                   ;mem1 =7ffe 0f7d 0ffa 8003
        psubusw     rmmx0, mem1 ;rmmx0=7ffd 12a5 2388 0000
...
```

Обратите внимание на последнюю шестнадцатеричную тетраду, которая соответствует результату вычитания (5–32771).

Команды умножения

Команды MMX-умножения предназначены только для умножения 16-битных элементов, причем реализация этих команд сделана несколько непривычно. Команды умножения целочисленного устройства формируют результат, размер которого вдвое превышает размер исходных операндов. Команды умножения MMX-расширения реализуют это действие несколько иначе:

- умножению подвергаются одновременно 4 слова со знаком;
- для получения полного результата умножения (размером в двойное слово) необходимо применение двух команд `pmulhw` и `pmullw`. С помощью этих команд формируются, соответственно, старшая и младшая части произведения. Для их объединения в единое двойное слово можно использовать команды расширения `punpckhwd` или `punpcklwd`.

`pmulhw приемник, источник` – умножение четырех знаковых упакованных слов. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* – либо MMX-регистр, либо 64-разрядная ячейка памяти. В *приемник* записываются не все 32 бита произведения, а только старшие 16 бит. Младшие 16 бит можно получить, используя команду `pmullw`.

`pmullw приемник, источник` – умножение знаковых упакованных слов. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* – либо MMX-регистр, либо 64-разрядная ячейка памяти. В *приемник* записываются не все 32 бита произведения, а только младшие 16 бит. Старшие 16 бит можно получить, используя команду `pmulhw`.

Для иллюстрации работы этих двух команд рассмотрим пример умножения четырех знаковых слов.

```
.data ;сегмент данных
memdw 5h
    df 7ffb22223382h
mem1 dw 8008h
    df 7ffe0f7d0ffah
mem2 dw 0
    df 0
mem3 dw 0
    df 0
.code
.....
    movq rmmx0, mem ;rmmx0=7ffb 2222 3382 0005
                      ;mem1 =7ffe 0f7d 0ffa 8008
;получим младшие части произведений
    pmullw rmmx0, mem1 ;rmmx0=800a ab9a eaf4 8028
    movq rmmx1, mem ;rmmx1=7ffb 2222 3382 0005
                      ;mem1 =7ffe 0f7d 0ffa 8003
;получим старшие части произведений
    pmulhw rmmx1, mem1 ;rmmx1=3ffc 0210 0336 fffd
;сохраним rmmx0 для последующих действий
    movq rmmx2, rmmx0 ;rmmx2=800a ab9a eaf4 8028
;в поле mem2 содержится результат произведений операндов младших половин исходных
операндов
    punpcklwd rmmx0, rmmx1 ;rmmx0=0336 eaf4 fffd 8028
    movq mem2, rmmx0 ;mem2=0336 eaf4 fffd 8028
;в поле mem3 полный результат произведений операндов старших половин исходных
операндов
    punpckhwd rmmx2, rmmx1 ;rmmx2=3ffc 800a 0210 a89a
    movq mem3, rmmx2 ;mem3=3ffc 800a 0210 a89a
....
```

С помощью программы-калькулятора Windows вы можете проверить полученные произведения. Вычислите с его помощью произведение $7ffb * 7ffe = 3fc 800a$. Такое же значение мы получили в поле `mem3`. Аналогично можно проверить и другие результаты.

`pmaddwd приемник, источник` — умножение четырех знаковых упакованных слов. Результат помещается в `приемник`, который является одним из MMX-регистров. `Источник` — либо MMX-регистр, либо 64-разрядная ячейка памяти. Формирование результата осуществляется по схеме, представленной на рис. 20.5.

Результат работы команды `pmaddwd` представляет собой сумму произведений двух старших и двух младших слов операндов `приемники источник`. Результат получается в виде двух двойных упакованных слов. Ниже показаны схема работы (рис. 20.5) и пример использования команды `pmaddwd`.

```
.data ;сегмент данных
memdw 5h
    df 7ffb22223382h
mem1 dw 8008h
    df 7ffe0f7d0ffah
```

```

mem2    dw    0
df      0
mem3    dw    0
df      0
.code
...
    movq  rmmx0, mem   ;rmmx0=7ffb 2222 3382 0005
                      ;mem1 =7ffe 0f7d 0ffa 8008
    pmaddwd   rmmx0, mem1 ;rmmx0=420d 28a4 0334 6b1c
...

```

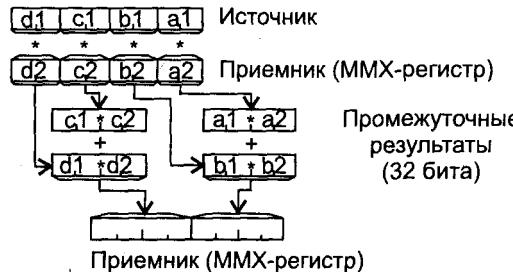


Рис. 20.5. Схема работы команды pmaddwd

Команды сравнения

Группа команд сравнения MMX-расширения содержит команды сравнения двух типов:

- простого сравнения — `pcmpqb` | `pcmpqw` | `pcmpqd`. Команды характеризуются тем, что устанавливают только факт равенства операндов — «равно-не равно»;
 - сравнения по величине — `cmpgtb` | `cmpgtw` | `cmpgtd`. Команды устанавливают соотношение операндов по величине.
- `pcmpqb` | `pcmpqw` | `pcmpqd` *операнд_1,операнд_2* — сравнение упакованных байтов, слов или двойных слов. Результат формируется в *операнд_1*, который является одним из MMX-регистров. *Операнд_2* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Элементы формируемого результата представляются в виде единичных или нулевых байтов, слов или двойных слов. Единичные байты, слова или двойные слова формируются, если соответствующие байты, слова или двойные слова исходных операндов равны. Нулевые байты, слова, двойные слова формируются в случае, если соответствующие байты, слова или двойные слова исходных операндов не равны.

```

.data ;сегмент данных
mem    dw    5h
       df    7fff22223382h
mem1   dw    5h
       df    7ffe0f7d0ffa
.code
...
    movq  rmmx0, mem   ;rmmx0=7fff 2222 3382 0005
                      ;mem1 =7ffe 0f7d 0ffa 0005

```

```
pcmpeqw    rmmx0, mem1 ;rmmx0=0000 0000 0000 ffff
```

`pcmpeqb | pcmpegtw | pcmpegtw` — сравнение по величине упакованных байтов, слов или двойных слов. Результат формируется в *операнд_1*, который является одним из MMX-регистров. *Операнд_2* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Элементы формируемого результата представляются в виде единичных или нулевых байтов, слов или двойных слов. Единичные байты, слова, двойные слова формируются в случае, если байты, слова или двойные слова исходного операнда *операнд_1* были больше соответствующих байтов, слов или двойных слов *операнд_2*. Иначе формируются нулевые байты, слова или двойные слова.

Пример ниже наглядно показывает, что единичные элементы получаются только в случае, если элементы первого операнда (всегда находящегося в MMX-регистре) больше соответствующих элементов второго операнда.

```
.data      ;сегмент данных
memdw    5h
df      7fff22223382h
mem1      dw    5h
df      7ffe0f7d0ffah
.code
...
    movq  rmmx0, mem   ;rmmx0=7fff 2222 3382 0005
                ;mem1 =7ffe 0f7d f0fa 0005
    pcmpegtw    rmmx0, mem1 ;rmmx0=ffff ffff 0000 0000
```

Команды логических операций

Команды логических операций предназначены для поразрядной обработки содержимого двух MMX-регистров или MMX-регистра и 64-битного операнда в памяти. Эти команды реализуют следующие логические операции: «И», «И-НЕ», «ИЛИ», «исключающее ИЛИ». Заметьте, что появилась новая логическая операция «И-НЕ», которой нет среди логических команд основного процессора. Другая особенность логических команд заключается в том, что они выполняются над всеми шестидесятью четырьмя разрядами MMX-операндов. Для структурирования результатов их работы нужно применять другие команды MMX-расширения.

`pand` *приемник,источник* — выполнение поразрядной операции «И» над операндами *приемник* *источник*. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в *приемник* формируются в соответствии с табл. 20.4.

Таблица 20.4. Таблица истинности для операции «И»

| | | | | |
|-----------|---|---|---|---|
| Приемник | 0 | 0 | 1 | 1 |
| Источник | 0 | 1 | 0 | 1 |
| Результат | 0 | 0 | 0 | 1 |

Принцип работы команды `pand` демонстрирует пример:

```
.data      ;сегмент данных
memdw    0505h
```

```

df    7ff002203080h
mem1 dw    0005h
df    7ff002000f80h
.code
...
    movq rmmx0, mem   ;rmmx0=7ff0 0220 3080 0505
                      ;mem1 =7ff0 0200 0f80 0005
    pand rmmx0, mem1 ;rmmx0=7ff0 0200 0080 0005
...

```

pandn приемник, источник — выполнение поразрядной операции «И-НЕ» над операндами *приемник* и *источник*. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в *приемник* формируются в соответствии с табл. 20.5. Из нее видно, что единичные биты в результате могут появиться только в одном случае, когда единичные биты второго операнда совпадают с нулевыми битами первого операнда. Команда позволяет определить положение единичных битов второго операнда, которым не соответствуют единичные биты первого операнда.

Таблица 20.5. Таблица истинности для операции «И-НЕ»

| | | | | |
|-----------|---|---|---|---|
| Приемник | 0 | 0 | 1 | 1 |
| Источник | 0 | 1 | 0 | 1 |
| Результат | 0 | 1 | 0 | 0 |

Принцип работы команды **pandn** демонстрирует пример:

```

.data      ;сегмент данных
memdw    0505h
df        7ff002203080h
mem1     dw    0005h
df        7ff003000f80h
.code
...
    movq rmmx0, mem   ;rmmx0=7ff0 0220 3080 0505
                      ;mem1 =7ff0 0300 0f80 0005
    pandn rmmx0, mem1 ;rmmx0=0000 0100 0f00 0000
...

```

por приемник, источник — выполнение поразрядной операции «ИЛИ» над операндами *приемник* и *источник*. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в *приемник* формируются в соответствии с табл. 20.6.

Таблица 20.6. Таблица истинности для операции «ИЛИ»

| | | | | |
|-----------|---|---|---|---|
| Приемник | 0 | 0 | 1 | 1 |
| Источник | 0 | 1 | 0 | 1 |
| Результат | 0 | 1 | 1 | 1 |

Принцип работы команды **por** демонстрирует пример:

```

.data ;сегмент данных
mem dw 0505h
      df 7ff002203080h
mem1 dw 0005h
      df 7ff003000f80h
.code

```

```

    movq rmmx0, mem ;rmmx0=7ff0 0220 3080 0505
                      ;mem1=7ff0 0300 0f80 0005
    por rmmx0, mem1 ;rmmx0=7ff0 0320 3f80 0505

```

pxor приемник, источник — выполнение поразрядной операции «исключающее ИЛИ» над operandами *приемники источник*. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в *приемник* формируются в соответствии с табл. 20.7.

Таблица 20.7. Таблица истинности для операции «исключающее ИЛИ»

| | | | | |
|-----------|---|---|---|---|
| Приемник | 0 | 0 | 1 | 1 |
| Источник | 0 | 1 | 0 | 1 |
| Результат | 0 | 1 | 1 | 0 |

Принцип работы команды pxor демонстрирует пример:

```

.data ;сегмент данных
memdw 0505h
      df 7ff002203080h
mem1 dw 0005h
      df 7ff003000f80h
.code

```

```

    movq rmmx0, mem ;rmmx0=7ff0 0220 3080 0505
                      ;mem1=7ff0 0300 0f80 0005
    pxor rmmx0, mem1 ;rmmx0=0000 0120 3f00 0500

```

Команды сдвига

На уроке 9 мы знакомились с командами арифметического и логического сдвига основного процессора. Отличие этих двух типов команд сдвига — в способе интерпретации знакового бита операнда. Среди MMX-команд сдвига также существуют команды арифметического и логического сдвига. Обратите внимание на то обстоятельство, что MMX-команд сдвига упакованных байтов нет. Сдвигать можно только упакованные слова, двойные слова и учетверенные слова (целиком весь MMX-регистр).

psllw | pslld | psllq приемник, источник — команды логического сдвига влево упакованных слов, двойных слов или учетверенных слов в *приемник* на количество разрядов, указанных значением *источник*. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига младшие биты упакованных элементов *приемник* заполняются нулями.

`psllw | pslld | psllq` *приемник, количество_сдвигов* — команды логического сдвига влево аналогичны рассмотренным выше командам, за исключением того, что все упакованные слова, двойные слова и учетверенные слова в *приемник* сдвигаются на количество разрядов, указанных значением непосредственного операнда *источник*. Освобождающиеся в результате сдвига младшие биты упакованных элементов *приемник* заполняются нулями.

`psrlw | psrld | psrlq` *приемник, источник* — команды логического сдвига вправо упакованных слов, двойных слов или учетверенных слов в *приемник* на количество разрядов, указанных значением в *источник*. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига старшие биты упакованных элементов *приемник* заполняются нулями.

`psrlw | psrld | psrlq` *приемник, количество_сдвигов* — команды логического сдвига вправо аналогичны рассмотренным выше командам, за исключением того, что все упакованные слова, двойные слова или учетверенные слова в *приемник* сдвигаются на количество разрядов, указанных значением непосредственного операнда *источник*. Освобождающиеся в результате сдвига старшие биты упакованных элементов *приемник* заполняются нулями.

```
.data ;сегмент данных
mem    dw      0fffffh
        df      0xfffffffffffffh
mem1   dw      4
        df      0
.code
        ...
        movq  rmmx0, mem  ;rmmx0=ffff ffff ffff ffff
                    ;mem1 =0000 0000 0000 0004
        psllw rmmx0, mem1 ;rmmx0=fff0 fff0 fff0 fff0
        psrlw rmmx0, 4     ;rmmx0=0fff 0fff 0fff 0fff
```

Следующие команды являются командами арифметического сдвига. Эти команды сдвигают значение операнда вправо. Команд арифметического сдвига влево нет, так как они аналогичны командам логического сдвига, не сохраняющим значения знакового разряда.

`psraw | psrad` *приемник, источник* — команды арифметического сдвига вправо упакованных слов или двойных слов в *приемник* на количество разрядов, указанных в *источник*. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига старшие биты упакованных элементов *приемник* заполняются значением знаковых (старших) разрядов этих элементов.

```
.data ;сегмент данных
mem    dw      0fff0h
        df      0fff0fff0fff0h
mem1   dw      4
        df      0
.code
        ...
```

```

movq rmmx0, mem ;rmmx0=ffff0 fff0 fff0 ffff
;mem1 =0000 0000 0000 0004
psraw rmmx0, 4 ;rmmx0=ffff ffff ffff ffff

```

Команды упаковки и распаковки

Команды этих групп предназначены для изменения размерности элементов операндов с учетом их значений. Команды упаковки позволяют уменьшить размерность элементов в два раза. При этом если значение сжимаемого элемента больше максимального допустимого значения, которое может содержаться в элементе меньшего размера, то результат формируется по принципу знакового насыщения.

packssdw приемник, источник — команда упаковки со знаковым насыщением двух двойных слов в *приемник* и двух двойных слов в *источник* в четыре слова в *приемник*. Схема выполнения команды показана на рис. 20.6. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти.



Рис. 20.6. Схема работы команды packssdw

packsswb приемник, источник — команда упаковки со знаковым насыщением четырех слов в *приемники* четырех слов в *источник* в четыре слова в *приемник*. Схема выполнения команды показана на рис. 20.7. Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти.

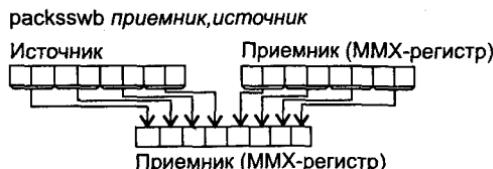


Рис. 20.7. Схема работы команды packsswb

```

.data      ;сегмент данных
memdw    0fe00h
df       00457ffe00f0h
.code
...
packsswb   rmmx0, mem ;rmmx0=45 7f 7f 80 00 00 00 00

```

Пример показывает, как выполняется принцип знакового насыщения результата до значений 7fh и 80h байт. Подобная ситуация возникает каждый раз, когда значение в исходном слове превышает максимально возможное значение.

Следующая группа MMX-команд позволяет выполнить обратную операцию — расширить размер элементов операнда в два раза. При этом недостающая половина вновь формируемого элемента извлекается из второго операнда.

р unpckhbw приемник, источник — команда распаковки байтов из старшей половины **приемник** в слова с использованием в качестве старшей половины этих слов байтов из **источник**. Формирование результата происходит путем поочередной выборки байтов из **приемник** и **источник** (рис. 20.8). Результат помещается в **приемник**, который является одним из MMX-регистров. **Источник** — либо MMX-регистр, либо 64-разрядная ячейка памяти.

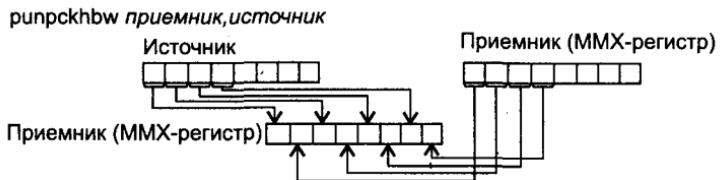


Рис. 20.8. Схема работы команды unpckhbw

```
.data ;сегмент данных
mem    dw      0
       df      01020304fffffh
mem1   dw      0
       df      0f0f0f0feeeeh
.code
... ...
    movq  rmmx0, mem   ;rmmx0=01 02 03 04 ff ff 00 00
                      ;mem1 =0f 0f 0f ee ee 00 00
    unpckhbw   rmmx0, mem1 :rmmx0=0f01 0f02 0f03 0f04
... ...
```

р unpckhwd приемник, источник — команда распаковки слов из старшей половины **приемник** в двойные слова с использованием в качестве старшей половины этих двойных слов из **источник**. Формирование результата происходит путем поочередной выборки слов из **приемник** и **источник** (рис. 20.9). Результат формируется в **приемник**, который является одним из MMX-регистров, в то же время **источник** может быть либо MMX-регистром, либо 64-разрядной ячейкой памяти.

р unpckhwd приемник, источник

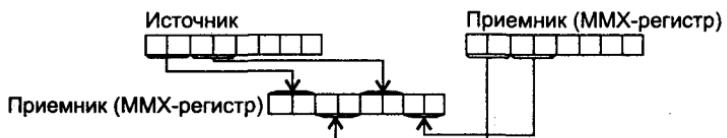


Рис. 20.9. Схема работы команды unpckhwd

р unpckhdq приемник, источник — команда распаковки двойных слов из старшей половины **приемник** в учетверенные слова с использованием в качестве старшей половины этих учетверенных слов двойных слов из **источник**. Формирование

результата происходит путем поочередной выборки двойных слов из *приемник* и *источник* (рис. 20.10). Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти.

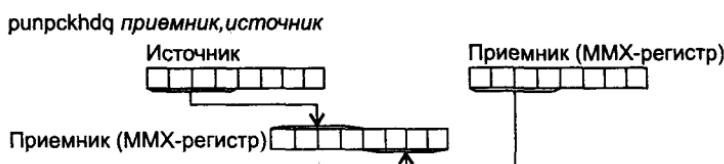


Рис. 20.10. Схема работы команды rinpckhdq

Вы, наверное, обратили внимание, что предыдущие три команды работают со старшими половинами операндов. Следующие три команды, наоборот, работают с младшими половинами операндов.

рипрcklbw *приемник, источник* — команда распаковки байтов из младшей половины *приемник* в слова с использованием в качестве младшей половины этих слов байтов из *источник*. Формирование результата осуществляется путем поочередной выборки байт из *приемник* и *источник* (рис. 20.11). Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — MMX-регистр, либо 64-разрядная ячейка памяти.

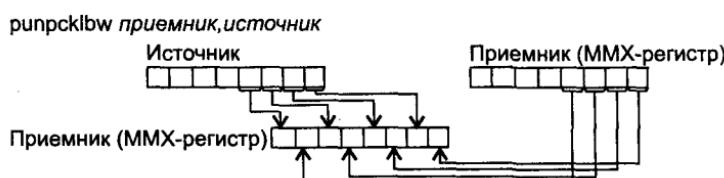


Рис. 20.11. Схема работы команды rinpcklbw

```
.data ;сегмент данных
mem dw 0304h
      df 0ffffeeee0102h
mem1 dw 0f0fh
      df 0c0c0c0c0f0fh
.code
...
    movq rmmx0, mem ;rmmx0=ff ff ee ee 01 02 03 04
                      ;mem1 =0c 0c 0c 0f 0f 0f 0f
    rinpcklbw rmmx0, mem1 ;rmmx0=0f01 0f02 0f03 0f04
...
```

рипрck1wd *приемник, источник* — команда распаковки слов из младшей половины *приемник* в двойные слова с использованием в качестве младшей половины этих двойных слов слов из *источник*. Формирование результата осуществляется путем поочередной выборки слов из *приемник* и *источник* (рис. 20.12). Результат помещается в *приемник*, который является одним из MMX-регистров, в то же время *источник* может быть либо MMX-регистром, либо 64-разрядной ячейкой памяти.

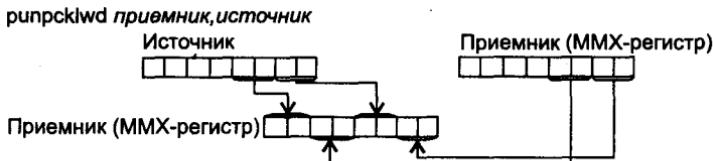


Рис. 20.12. Схема работы команды ruprcklwd

ruprckldq приемник, источник — команда распаковки двойных слов из младшей половины *приемника* в учетверенные слова с использованием в качестве младшей половины этих учетверенных слов двойных слов из *источника*. Формирование результата осуществляется путем поочередной выборки двойных слов из *приемника* и *источника* (рис. 20.13). Результат помещается в *приемник*, который является одним из MMX-регистров. *Источник* — либо MMX-регистр, либо 64-разрядная ячейка памяти.

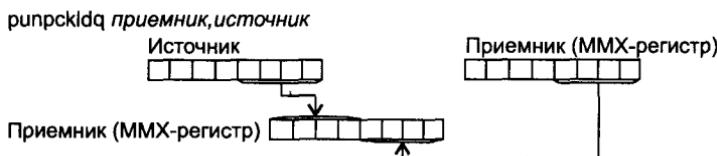


Рис. 20.13. Схема работы команды ruprckldq

Команда очистки стека регистров сопроцессора

В конце блока команд, содержащего MMX-инструкции, должна обязательно присутствовать команда **emms**, в функции которой входит очистка стека регистров и установка единичных значение в регистре тегов. Каждому регистру стека сопроцессора соответствует двухбитовое поле в этом регистре, единичное значение которого говорит о том, что регистр пуст.

emms — команда очистки MMX-контекста. Не имеет операндов.

Наличие этой команды обязательно, если MMX-команды комбинируются с командами сопроцессора.

В настоящее время пока еще достаточно большое количество компьютеров работает на микропроцессорах, не поддерживающих MMX-технологию. С тем чтобы количество пользователей вашей программы было как можно большим, это обстоятельство необходимо учитывать. При разработке программы необходимо предусматривать альтернативные участки кода, на которых производится эмуляция работы MMX-команд с использованием целочисленных команд. Перед началом работы такая программа должна выполнять проверку текущего микропроцессора на возможность поддержки им MMX-команд. По результатам этой проверки управление передается на соответствующий участок кода. На основе чего выполняется подобная проверка?

В систему команд микропроцессоров, начиная с i586 (Pentium), входит команда **cpu id**, с помощью которой можно получить информацию о текущем микропроцессоре. Листинг 20.6 содержит пример программы, которая определяет факт поддержки микропроцессором MMX-технологии.

Листинг 20.6 Определение поддержки микропроцессором MMX-технологии

```
.586p
model use16 small
.stack 100h
.data ;сегмент данных
mmx_mes db "Микропроцессор поддерживает MMX-технологию", "$"
no_mmx_mes db "Микропроцессор не поддерживает MMX-технологию", "$"
.code
main proc ;начало процедуры main
    mov ax, @data
    mov ds, ax
    xor edx, edx
    mov eax, 1
    cpuid
    bt edx, 23
    jnc no_mmx
    mov ah, 9
    mov dx, offset mmx_mes
    int 21h
    jmp exit
no_mmx: mov ah, 9
    mov dx, offset no_mmx_mes
    int 21h
exit:
    mov ax, 4c00h ;пересылка 4c00h в регистр ах
    int 21h ;вызов прерывания с номером 21h
main endp      ;конец процедуры main
end main       ;конец программы с точкой входа main
```

Пример применения MMX-технологии

Как правило, мультимедийный файл представляет собой массив однородных элементов. Этому массиву предшествует некоторая описательная информация (заголовок), в котором содержится общая информация о файле. Так как в подобных массивах размер элементов обычно одинаков, то их удобно обрабатывать группами. Наибольшая эффективность работы программы достигается, если использовать MMX-команды.

В качестве типового мультимедийного файла можно рассматривать файл растрового изображения. Существует несколько наиболее часто используемых файлов этого типа. Наиболее простую структуру имеют bmp-файлы. Рассмотрим типовое применение MMX-команд на примере обработки файлов этого типа. Возможно, читатель в своей работе сталкивался с необходимостью преобразования цветного изображения в полутоновое. Подобную возможность, в частности, поддерживают многие популярные редакторы растровой графики. Попробуем самостоятельно решить эту задачу, используя возможности команд MMX-расширения.

Структурно файл растрового изображения в формате .bmp состоит минимум из двух, и максимум из трех блоков:

1. Обязательный заголовок с информацией, характеризующий растровое изображение. По отношению к содержащейся в этом заголовке информации, его можно разделить на две части:

- первая часть, имеющая размер 14 байтов, — предназначена для идентификации файла как растрового графического и хранения общей информации о нем;
 - вторая часть, имеющая размер 40 байтов, — содержит характеристики самого растрового изображения.
2. Необязательный массив с информацией о цвете. Элементы этого массива представляют собой структуру, содержащую четыре поля. Размерность каждого поля составляет один байт. Три поля этой структуры содержат интенсивности синего, красного и зеленого цветов (RGB-модель цвета), четвертое поле всегда равно нулю и предназначено для выравнивания начала следующего элемента на четную границу. Для микропроцессоров архитектуры Intel это означает существенное повышение производительности, так как выровненные данные выбираются из памяти быстрее. Более того, это обстоятельство дает хорошие предпосылки к эффективному использованию команд MMX-расширения, так как данные во время обработки можно группировать. Массив с информацией о цвете отсутствует для растровых изображений с 24-битным представлением цвета (True Color). В случае с нашей программой мы имеем на входе именно такой файл. На выходе мы должны будем получить файл с полутоновым изображением, который уже будет содержать подобный массив цветов.
3. Обязательный массив с описанием пикселов растрового изображения. Формат элементов этого массива зависит от типа растрового изображения. В нашем случае мы имеем два типа растрового изображения: 24-битное (True Color) и 8-битное (полутоновое). Как отмечено выше, файл с описанием 24-битного изображения не содержит массива с информацией о цвете. В этом файле непосредственно за заголовком (54 бита) следует информация о цвете пикселов. Каждый пиксель описывается тремя байтами, которые содержат значения (из диапазона 0...255) интенсивностей красного, зеленого и красного цвета. Для других типов изображений информация о цвете каждого пикселя предоставляется по другому принципу. К примеру, для восьмибитного изображения (которое мы должны сформировать в нашей задаче) информация о цвете конкретного пикселя получается следующим образом. Каждый пиксель в массиве с описанием пикселов растрового изображения описывается одним байтом. Значение этого байта является индексом в массиве с информацией о цвете, рассмотренном в предыдущем пункте. Соответствующий элемент массива с информацией о цвете содержит значения красной, синей и зеленой составляющих цвета пикселя.

Рассмотрим суть алгоритма обработки, который реализует программа преобразования растрового изображения (листинг 20.7). На рисунках 20.14 и 20.15 приведены фрагменты исходного bmp-файла (24-битное изображение) и соответствующего выходного bmp-файла (8-битное полутоновое изображение), сформированного в результате обработки программой листинга 20.4.

На рис. 20.15 с адреса 0000:0436 и до конца файла (адрес 0002:F875) располагается описание пикселов. Каждый байт в этом описании является индексом массива цветовых значений.

Программа преобразования 24-битного растрового изображения в 8-битное выполнена в виде консольного приложения Windows. Таким образом, мы решили сразу две задачи: во-первых, показали практический пример использования воз-

можностей MMX-технологии обработки данных и, во-вторых, показали порядок разработки на языке ассемблера еще одного типа приложений Windows — консольного.

| | | |
|-----------|---|------------------|
| 0000:0000 | 42 4D F6 DC 08 00 00 00 00 00 36 00 00 00 28 00 | BM.....6...(. |
| 0000:0010 | 00 00 B8 01 00 00 B8 01 00 00 01 00 18 00 00 00 | |
| 0000:0020 | 00 00 C0 DC 08 00 C2 1E 00 00 C2 1E 00 00 00 00 | |
| 0000:0030 | 00 00 00 00 00 00 49 7A A6 4A 7B A7 4D 7E AA 4F |Iz.J{.M-.O |
| 0000:0040 | 80 AC 50 81 AD 50 81 AD 4F 80 AC 4F 80 AC 4B 7C | þ.P..P..0þ.0þ.K |
| ... | | |
| 0008:DCE0 | A5 7A 93 A3 79 93 A3 79 92 A6 7D 99 B1 92 AD C8 | .z...y..y..}.... |
| 0008:DCF0 | AD CA E5 C2 DE FC | |

Рис. 20.14. Фрагмент файла 24-битного растрового изображения в формате bmp

На рис. 20.14 с адреса 0000:0036 и до конца файла (адрес 0008:DCF5) располагается описание пикселов — три последовательных байта описывают красную, синюю и зеленую составляющие одного пикселя.

| | | |
|-----------|--|-------------------|
| 0000:0000 | 42 4D 76 F8 02 00 00 00 00 00 36 04 00 00 28 00 | BMv.....6...(. |
| 0000:0010 | 00 00 B8 01 00 00 B8 01 00 00 01 00 08 00 00 00 | |
| 0000:0020 | 00 00 40 F4 02 00 C2 1E 00 00 C2 1E 00 00 00 01 | ..@..... |
| 0000:0030 | 00 00 00 00 00 00 00 00 00 00 00 01 01 01 00 02 02 | |
| 0000:0040 | 02 00 03 03 03 00 04 04 04 00 05 05 05 00 06 06 | |
| 0000:0050 | 06 00 07 07 07 00 08 08 08 00 09 09 09 00 0A 0A | |
| 0000:0060 | 0A 00 0B 0B 0B 00 0C 0C 0C 00 0D 0D 0D 00 0E 0E | |
| 0000:0070 | 0E 00 0F 0F 0F 00 10 10 10 00 11 11 11 00 12 12 | |
| 0000:0080 | 12 00 13 13 13 00 14 14 14 00 15 15 15 00 16 16 | |
| 0000:0090 | 16 00 17 17 17 00 18 18 18 00 19 19 19 00 1A 1A | |
| 0000:00A0 | 1A 00 1B 1B 1B 00 1C 1C 1C 00 1D 1D 1D 00 1E 1E | |
| 0000:00B0 | 1E 00 1F 1F 1F 00 20 20 20 00 21 21 21 00 22 22 |!!!."" |
| 0000:00C0 | 22 00 23 23 23 00 24 24 24 00 25 25 25 00 26 26 | ".###.###.###.%&. |
| ... | массив цветовых значений | |
| 0000:0400 | F2 00 F3 F3 00 F4 F4 00 F5 F5 00 F6 F6 | |
| 0000:0410 | F6 00 F7 F7 00 F8 F8 00 F9 F9 00 FA FA | |
| 0000:0420 | FA 00 FB FB 00 FC FC 00 FD FD 00 FE FE | |
| 0000:0430 | FE 00 FF FF FF 00 70 71 74 76 77 77 76 76 72 75 |pqtvwwvvru |
| 0000:0440 | 78 77 72 69 5F 5A 61 62 63 66 69 6D 72 74 67 70 | xwri_Zabcfimrtgp |
| 0000:0450 | 77 77 74 72 70 6D 6E 6B 69 6A 6C 6C 68 64 63 6F | wwtrpmnkij11hdco |
| 0000:0460 | 74 6F 6F 76 7A 75 68 73 75 6F 74 7E 78 69 76 71 | toovzuhsuot~xivq |
| 0000:0470 | 6A 68 6C 74 7A 7E 86 7B 78 7C 78 6A 67 6E 69 79 | jhltz~.{x xjgniy |
| ... | | |
| 0002:F860 | 73 71 6D 6C 70 74 77 75 75 79 7D 7E 7C 79 8B 8D | sqm1ptwuuuy}~ y.. |
| 0002:F870 | 8C 8C 93 A7 C4 D8 | |

Рис. 20.15. Фрагмент файла 8-битного растрового изображения в формате bmp

Программа преобразования (листинг 20.7) последовательно выполняет следующие действия:

О вводит имена исходного и выходного файла. Следует отметить следующее требование к исходному файлу. Этот файл должен быть отсканирован с разрешением не менее 200 dpi, в противном случае возможны искажения. В нашем случае борьба за качество не является основной задачей, поэтому для

исправления возможных искажений в программе не предпринимается никаких действий. Если сканера у вас нет, то нужно подобрать файл соответствующего качества. Если у вас ни того, ни другого нет, то на дискеете, прилагаемой к книге, есть файл, с которым вы можете проводить необходимые эксперименты;

- открывает исходный и создается выходной файлы. Для работы с этими файлами используется еще один полезный механизм Windows – *файлы, проецируемые в память*. Этот механизм позволяет открыть файлы специальным образом и работать с ними далее как с обычными массивами. При рассмотрении программы обратите внимание на эту возможность и то, как ею пользоваться в программах на языке ассемблера;
- определяет корректность исходного файла. При этом проверяется два обстоятельства: первое – является ли он bmp-файлом, и, второе – является ли он 24-битным файлом растрового изображения;
- формирует заголовок выходного файла на основании информации исходного файла. Основными при этом являются две задачи – определение размера выходного файла и заполнение полей, определяющих особенности данного растрового файла;
- формирует массив цветов. Он содержит 256 элементов, что позволяет формировать 256-цветное растровое изображение. Файл с 8-битным полутонаовым растровым изображением структурно ничем не отличается от 8-битного цветного файла. Единственное отличие можно увидеть при сравнении массивов цветов. Фрагмент этого массива для 8-битного полутонаового изображения показан на рис. 20.24. Хорошо видно, что в каждом элементе этого массива значения красной, зеленой и синей составляющей одинаковы. Это особенность формирования полутонаового изображения. Для цветных изображений подобной закономерности не существует, так как оттенки цветного изображения формируются комбинациями отличающихся значений красной, зеленой и синей составляющих;
- формирует значения пикселов. Как мы уже упоминали выше, каждый пиксель описывается одним байтом, который на самом деле является лишь индексом, обозначающим цвет в массиве цветов;
- закрывает файлы. В конце текста программы необходимо, с помощью соответствующих функций Windows, корректно закрыть файлы, в противном случае результаты работы программы будут потеряны.

Более детально алгоритм работы программы поясним по ходу рассмотрения ее текста в листинге 20.7. При этом мы не будем обсуждать функций Windows, обеспечивающих функционирование программы. Эту информацию вы можете почерпнуть из многочисленных источников. Основное внимание будет уделено использованию MMX-команд.

Листинг 20.7. Преобразование цветного изображения в полутонаовое

```
:prg21c.asm
;Пример консольного приложения для Win32
;(с использованием команд MMX-расширения и файлов, проецируемых в память)
.486
.model flat, STDCALL ;модель памяти flat
;STDCALL – передача параметров в стиле С (справа налево)
```

```
; вызываемая процедура чистит за собой стек  
%NOINCL ;запретить вывод текста включаемых файлов  
include mmx32.inc  
include WindowConA.inc  
;Объявление внешними используемых в данной программе функций Win32 (ASCII):  
extrn AllocConsole:PROC  
extrn SetConsoleTitleA:PROC  
extrn ExitProcess:PROC  
extrn GetStdHandle:PROC  
extrn CreateFileA:PROC  
extrn CreateFileMappingA:PROC  
extrn MapViewOfFile:PROC  
extrn UnmapViewOfFile:PROC  
extrn CloseHandle:PROC  
extrn FlushFileBuffers:PROC  
extrn FlushViewOfFile:PROC  
extrn WriteFile:PROC  
extrn SetConsoleCursorPosition:PROC  
extrn ReadConsoleA:PROC  
extrn WriteConsoleA:PROC
```

;структура bmp-файла

;макроопределения типов

```
SSHORT equ <dw 0>  
UINT equ <dw 0>  
DDWORD equ <dd 0>  
LONG equ <dd 0>  
WWORD equ <dw 0>  
BYTE equ <db 0>
```

;структура для установки положения курсора в консоли:

```
Coord struc
```

```
xx SSHORT
```

```
yy SSHORT
```

```
Coord ends
```

;заголовок bmp-файла

```
BitMapFileHeader struc
```

```
bfType UINT ;символы «B» и «M»
```

```
bfSize DDWORD ;размер файла в байтах
```

```
bfReserved1 UINT ;резерв
```

```
bfReserved2 UINT ;резерв
```

```
bfOffBits DDWORD ;смещение в байтах к началу растрового изображения
```

;характеристики растрового изображения

```
biSize DDWORD ;размер данной структуры в байтах (должно быть равно 00000028h)
```

```
biWidth LONG ;ширина изображения в пикселях
```

```
biHeight LONG ;высота изображения в пикселях
```

```
biPlanes WWORD ;число цветовых плоскостей (должно быть равно 1)
```

```
biBitCount WWORD ;число бит на пиксел (должно быть равно 1, 4, 8, 24)
```

```
biCompression DDWORD ;метод сжатия
```

```
biSizeImage DDWORD ;размер собственно растрового изображения в байтах
```

```
biXPelsPerMeter LONG ;разрешение по горизонтали в пикселях на метр
```

```
biYPelsPerMeter LONG ;разрешение по вертикали в пикселях на метр
```

```
biClrUsed DDWORD ;число цветов в изображении
```

```
biClrImportant DDWORD ;число важных цветов изображения
```

```
BitMapFileHeader ends
```

```
.data
```

Листинг 20.7 (продолжение)

```
NumWri    dd    0
inFile    db    80 dup (20)
outFile   db    80 dup (20)
conCoord  <>
hinFile   dd    0
houtFile  dd    0
hMapinFile dd    0
hMapoutFile dd    0
TitleText db    'MMX-преобразование bmp-файла', 0
mes1      db    'Введите путь к исходному файлу TrueColor:'
len_mes1=$-mes1
mes2      db    'Введите путь к выходному файлу :'
len_mes2=$-mes2
mesErr1   db    'Это не bmp-файл:'
len_mesErr1=$-mesErr1
mesErr2   db    'Это не TrueColor-файл:'
len_mesErr2=$-mesErr2
mesRet    db    'Нажмите любую клавишу для выхода'
len_mesRet=$-mesRet
dOut      dd    0
dIndd    0
porog    label dword
        dw    0000h
        dw    004dh
        dw    0097h
        dw    001ch :00 77 151 28
Rab0b1   dd    0
        dd    0
pixSizeLow dd    0
pixSizeHi dd    0
i8 dd    8
outFileSize dd    0
initRMMX0 label dword
        db    00, 00, 00, 00, 01, 01, 01, 00
initRMMX1 label dword
        db    02, 02, 02, 00, 02, 02, 02, 00
PointOutRegion dd    0
PointInRegion dd    0
nWrByte   dd    0
temp      db    "В"
.code
start     proc near
;точка входа в программу:
;запрос консоли
        call AllocConsole
;проверить успех запроса консоли
        test ax, ax
        jz exit ;неудача
;текст окна заголовка
        push offset TitleText
        call SetConsoleTitleA
;вывод строки текста
```

```
;вначале получим дескрипторы ввода и вывода консоли
push STD_OUTPUT_HANDLE
call GetStdHandle
mov dOut, eax ;dOut-дескриптор ввода-вывода консоли
push STD_INPUT_HANDLE
call GetStdHandle
mov dIn, eax ;dIn-дескриптор ввода-вывода консоли
;установим курсор в позицию (2, 5)
mov con.xx, 2
mov con.yy, 5
push con
push dOut
call SetConsoleCursorPosition
cmp eax, 0
jz exit ;если неуспех
;вывести приглашение на ввод имени исходного файла
push 0
push offset NumWri
push len_mes1
push offset mes1
push dOut
call WriteConsoleA
;установим курсор в позицию (2, 6)
mov con.xx, 2
mov con.yy, 6
push con
push dOut
call SetConsoleCursorPosition
cmp eax, 0
jz exit ;если неуспех
push 0
push offset NumWri
push 80
push offset inFile
push dIn
call ReadConsoleA
lea eax, inFile
sub eax, NumWri, 2
add eax, NumWri
mov [eax], byte ptr 0
;установим курсор в позицию (2, 7)
mov con.xx, 2
mov con.yy, 7
push con
push dOut
call SetConsoleCursorPosition
cmp eax, 0
jz exit ;если неуспех
;вывести приглашение на ввод имени выходного файла
push 0
push offset NumWri
push len_mes2
push offset mes2
push dOut
```

```

call WriteConsoleA
;установим курсор в позицию (2, 8)
mov con.xx, 2
mov con.yy, 8
push con
push dOut
call SetConsoleCursorPosition
cmp eax, 0
jz exit ;если неуспех
push 0
push offset NumWri
push 80
push offset outFile
push dIn
call ReadConsoleA
lea eax, outFile
sub NumWri, 2
add eax, NumWri
mov [eax], byte ptr 0
;----- inFile -----
;открытие объекта ядра "файл" для исходного файла inFile
push NULL
push FILE_ATTRIBUTE_NORMAL
push OPEN_ALWAYS
push NULL
push 0
push GENERIC_READ+GENERIC_WRITE ;разрешено чтение и запись в файл
push offset inFile
call CreateFileA
;проверить успех
cmp eax, 0xffffffff
jz exit ;неудача
mov hinFile, eax
;создание объекта ядра «проецируемый файл» для исходного файла inFile
push NULL
push 0 ;размер файла – текущий
push 0
push PAGE_READWRITE
push NULL
push hinFile
call CreateFileMappingA
;проверить успех
cmp eax, 0
jz exit ;неудача
mov hMapinFile, eax
;проецирование файловых данных для исходного файла inFile на адресное
;пространство процесса
push NULL
push 0 ;смещение первого считываемого байта в файле
push 0
push FILE_MAP_WRITE
push hMapinFile

```

```
call MapViewOfFile
;проверить успех открытия файла
cmp eax, 0
jz exit ;неудача
mov PointInRegion, eax
mov ebx, eax ;адрес начала исходного файла в памяти в ebx
;----- inFile -----
mov ebx, eax ;адрес файла в памяти в ebx
cmp [ebx].biSize, 28h
jne exit_err1 ;это не bmp-файл
cmp [ebx].biBitCount, 18h
jne exit_err2 ;это не TrueColor файл
;преобразование True -> Gray
;----- outFile -----
;открытие объекта ядра "файл" для выходного файла outFile
push NULL
push FILE_ATTRIBUTE_NORMAL
push OPEN_ALWAYS
push NULL
push 0
push GENERIC_READ+GENERIC_WRITE ;разрешено чтение и запись в файл
push offset outFile
call CreateFileA
;проверить успех
cmp eax, 0xffffffff
jz exit ;неудача
mov houtFile, eax
;создание объекта ядра «проецируемый файл» для выходного файла outFile
;вычисляем длину выходного файла TrueColor в пикселях
;(умножаем длину изображения на его высоту (в пикселях))
    mov eax, [ebx].biWidth
    mul dword ptr [ebx].biHeight
    mov pixSizeLow, eax ;младшая половина размера раstra в пикселях
    mov pixSizeHi, edx ;старшая половина размера раstra в пикселях
;в eax размер собственно растрового изображения в байтах
    add eax, 54+4*256 ;54 - размер фиксированного заголовка
                      ;4*256 - размер массива цветовых значений
;в eax размер выходного файла в байтах
    mov outFileSize, eax
    push NULL
    push eax ;размер выходного файла
    push NULL
    push PAGE_READWRITE ;PAGE_WRITECOPY
    push NULL
    push houtFile
    call CreateFileMappingA
;проверить успех
    cmp eax, 0
    jz exit ;неудача
    mov hMapoutFile, eax
;проецирование файловых данных для выходного файла outFile на адресное
;пространство процесса
    push outFileSize
    push 0 ;смещение первого считываемого байта в файле
```

Листинг 20.7 (продолжение)

```
push 0
push FILE_MAP_WRITE      ;FileMapAllAccess
push hMapoutFile
call MapViewOfFile
;проверить успех открытия файла
cmp eax, 0
jz exit ;неудача
mov PointOutRegion, eax
mov edi, eax      ;адрес начала выходного файла в памяти в edi
```

Действиями выше мы подготовили приложение к выполнению собственно преобразования. Для этого средствами Windows для работы с консолью были введены имена файлов. После этого эти файлы были открыты как файлы, отображаемые в память. Адреса этих файлов в памяти были помещены в ячейки PointInRegion и PointOutRegion. Для удобства работы эти адреса содержатся также в регистрах ebx и edi, соответственно. Для того чтобы иметь возможность просматривать выходной файл стандартными средствами (графическим редактором Paint), необходимо сформировать корректный заголовок для выходного файла. При этом следует обратить внимание на формирование полей, характеризующих общий размер файла и размер собственно растрового изображения. Методика здесь простая. Из заголовка исходного файла извлекаются значения высоты и ширины раstra в пикселях. Их произведение и дает размер всего раstra в пикселях. Для 8-битного изображения это значение представляет собой длину раstra в байтах. Для получения размера всего файла остается прибавить размер массива цветов (256*4 байт) и размер заголовка bmp-файла (54 байта). Эти действия мы выполняли чуть выше, когда создавали выходной файл и определяли его размер. Результат этих действий мы будем использовать ниже для формирования соответствующих полей заголовка.

Листинг 20.7 (продолжение)

```
;формируем заголовок выходного файла:
```

```
mov ax, [ebx].bfType
mov [edi].bfType, ax
mov eax, outFileSize
mov [edi].bfSize, eax ;общий размер файла
mov [edi].bfReserved1, 0
mov [edi].bfReserved2, 0
mov [edi].bfOffBits, 54+4*256
mov [edi].biSize, 28h
mov eax, [ebx].biWidth
mov [edi].biWidth, eax ;ширина раstra в пикселях
mov eax, [ebx].biHeight
mov [edi].biHeight, eax      ;высота раstra в пикселях
mov [edi].biPlanes, 1 ;число цветовых плоскостей
mov [edi].biBitCount, 8      ;число бит на пиксель
mov [edi].biCompression, 0 ;метод сжатия
mov eax, outFileSize
sub eax, 54+4*256
mov [edi].biSizeImage, eax ;размер собственно растрового изображения
mov eax, [ebx].biXPelsPerMeter
mov [edi].biXPelsPerMeter, eax      ;ширина раstra в пикселях
```

```
mov    eax, [ebx].biYPelsPerMeter
mov    [edi].biYPelsPerMeter, eax      ;высота раstra в пикселях
mov    [edi].biClrUsed, 100h          ;число цветов в изображении
mov    [edi].biClrImportant, 0        ;число важных цветов в изображении
```

Далее формируем массив цветов. Особенности его содержимого мы обсуждали выше. Теперь нас интересуют особенности процесса его формирования. Каждый элемент этого массива занимает 4 байта, поэтому для его формирования удобно использовать MMX-команды. Содержание исполняемых действий очень простое, поэтому попробуйте разобраться в них самостоятельно, используя для справки описание команд MMX-расширения, приведенные в тексте занятия и в Справочнике.

Листинг 20.7 (продолжение)

```
;формируем массив цветов
add   edi, 54
mov   ecx, 128
movq  rmmx0, initRMMX0
movq  rmmx1, initRMMX1
xor   esi, esi
m3:
    movq  [edi+esi*8], rmmx0
    paddusb rmmx0, rmmx1
    dec   ecx
    inc   esi
    jcxz  m1
    jmp   m3
```

После формирования массива цветов все готово для выполнения собственно преобразования растрового изображения. Оно выполняется по определенному алгоритму, суть которого заключается в следующем. Каждый пиксель цветного изображения преобразуется в эквивалентный оттенок серого цвета. Для получения такого оттенка необходимо красную, синюю и зеленую составляющие цвета исходного пикселя умножить на определенные коэффициенты, после чего сложить. Полученный результат делится на 256 (можно сдвигом на 8 разрядов вправо). Полученное в результате деления частное (размером в один байт) и является эквивалентным оттенком серого цвета, полученным из RGB-компонент исходного цветного пикселя. Схематично процесс преобразования цвета пикселя показан на рис. 20.16.

Листинг 20.7 (продолжение)

```
----- outFile -----
m1:
    mov   ecx, pixSizeLow
    add   edi, 4*256  ;[edi] - на начало собственно растрового изображения
                      ;в выходном файле
    add   ebx, [ebx]+0ah
    dec   ebx  ;в ebx адрес в файле начала изображения - 1
    xor   esi, esi
;zагрузка в регистры цветовых значений для преобразования
m2:
    punpcklbw rmmx0, [ebx][esi*4]
    psrlw rmmx0, 8
```

Листинг 20.7 (продолжение)

```
pmullw rmmx0, porog
movq rmmx1, rmmx0
psl1q rmmx1, 16
paddusw rmmx0, rmmx1
psl1q rmmx1, 16
paddusw rmmx0, rmmx1
psrlw rmmx0, 8      ;делим на 256
movq Rab0b1, rmmx0
mov al, byte ptr Rab0b1+6
mov [edi], al
add ebx, 3
dec ecx
inc edi
jecxz m5
jmp m2
m5: jmp exit
```

В данном варианте программы обрабатываются только две ошибочные ситуации: ввод некорректного имени файла (или пути к нему) и выбор в качестве исходного файла, файла с типом, отличным от TrueColor. В случае возникновения других ошибочных ситуаций, производится переход на метку `exit` с последующим выходом из программы.

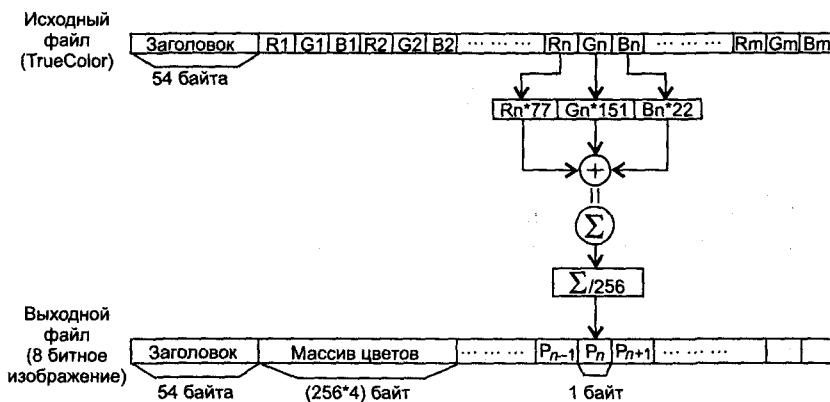


Рис. 20.16. Схема преобразования цветного пикселя (True Color) в эквивалентный серый

Листинг 20.7 (продолжение)

```
exit_err1:
;установим курсор в позицию (5, 10)
    mov    con.xx, 5
    mov    con.yy, 10
    push   con
    push   dOut
    call   SetConsoleCursorPosition
    cmp    eax, 0
    jz    exit ;если неуспех
;вывести сообщение об ошибке
```

```

push 0
push offset NumWri
push len_mes2
push offset mesErr1
push dOut
call WriteConsoleA
jmp exit ;return
exit_err2:
;установим курсор в позицию (5, 10)
mov con.xx, 5
mov con.yy, 10
push con
push dOut
call SetConsoleCursorPosition
cmp eax, 0
jz exit ;если неуспех
;вывести сообщение об ошибке
push 0
push offset NumWri
push len_mes2
push offset mesErr2
push dOut
call WriteConsoleA
: jmp return
;закрываем файлы
exit:
    emms
;установим курсор в позицию (5, 12)
mov con.xx, 5
mov con.yy, 10
push con
push dOut
call SetConsoleCursorPosition
;вывести сообщение о нажатии любой клавиши
push 0
push offset NumWri
push len_mesRet
push offset mesRet
push dOut
call WriteConsoleA
push 0
push offset NumWri
push 80
push offset inFile
push dIn
call ReadConsoleA

```

Перед выходом из программы необходимо должным образом закрыть файлы. Если этого не сделать, то их содержимое не изменится. Для корректного закрытия файла, спроектированного в память, необходимо вызвать функции `FlushViewOfFile` и `CloseHandle`. С помощью функции `FlushViewOfFile` измененные данные из памяти перекачиваются на диск. Функция `CloseHandle` вызывается дважды для закрытия объектов ядра «файл» и «проецируемый файл».

```

push NULL
push PointInRegion
call FlushViewOfFile
push hInFile
call CloseHandle
push hMapInFile
call CloseHandle
push outFileSize
push PointOutRegion
call FlushViewOfFile
push hOutFile
call CloseHandle
push hMapOutFile
call CloseHandle
;выход из приложения
return:
;готовим вызов VOID ExitProcess(UINT uExitCode)
push 0
call ExitProcess
start endp
endstart

```

Дополнительные целочисленные MMX-команды (Pentium III)

Эти команды нужно рассматривать как дополнение к системе целочисленных MMX-команд. Впервые они появились в архитектуре микропроцессора Pentium III. Общее их количество — 12.

ravgb | ravgw приемник, источник — команда вычисления среднего двух значений. Значения представляют собой беззнаковые целочисленные упакованные элементы операндов *приемник* и *источник* размером байт/слово. *Источник* может быть MMX-регистром или 64-битной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение *приемника*, который должен быть MMX-регистром. В нем формируется результат вычисления среднего арифметического. Сама операция выполняется над парными элементами в обоих операндах следующим образом:

- выполнить беззнаковое сложение парных элементов операндов *приемник* и *источник*;
- запомнить перенос в старшие разряды;
- сдвинуть вправо на один разряд результат сложения (без учета бита переноса), то есть разделить на 2;
- результат сдвига сложить со значением переноса.

Дополнительные MMX-команды придают MMX-расширению ряд новых возможностей для организации более эффективной обработки данных. Так, следующие рассматриваемые команды позволяют организовать доступ к отдельным элементам MMX-операнда.

pxextrw приемник, источник, маска — команда извлечения одного из четырех упакованных слов операнда *источник*. *Источник* должен быть MMX-регистром. Извлекаемое из *источника* слово локализуется с помощью значения, задаваемого маской. Значение в ней имеют два младших бита, которые численно определяют номер слова (от 0 до 3), извлекаемого из *источника*. Извлеченое слово помещается в младшее слово *приемника*, являющегося одним из 32-разрядных регистров общего назначения. Старшее слово *приемника* обнуляется;

pinsrw приемник, источник, маска — команда вставки слова в одно из четырех упакованных слов операнда *приемник*. *Приемник* должен быть MMX-регистром. *Источник* может быть 32-битным регистром общего назначения, или словом памяти. Место вставки слова локализуется в *приемнике* с помощью значения, задаваемого *маской*. Значение в ней имеют два младших бита, которые численно определяют номер слова (от 0 до 3) в приемнике, которое будет замещаться новым значением из *источника*. Если *источник* — 32-битный регистр, то вставляемое из него слово должно быть младшим;

rmaxhb | rmaxsw приемник, источник — команда извлечения максимального значения из каждой пары упакованных элементов в операндах *источник* и *приемник*. Элементы представляют собой беззнаковые байты (для команды *rmaxhb*) или знаковые слова (для команды *rmaxsw*). *Приемник* должен быть MMX-регистром. *Источник* может быть MMX-регистром или 64-битной ячейкой памяти. Результат из максимальных элементов каждой пары формируется в операнде *приемник*;

rmminhb | rmminsw приемник, источник — команда для извлечения минимального значения из каждой пары упакованных элементов в операндах *источник* и *приемник*. Элементы представляют собой беззнаковые байты (для команды *rmminhb*) или знаковые слова (для команды *rmminsw*). *Приемник* должен быть MMX-регистром. *Источник* может быть MMX-регистром или 64-битной ячейкой памяти. Результат из минимальных элементов каждой пары формируется в операнде *приемник*.

Следующая команда необычная — она позволяет выделить значения знака (старшего бита) упакованных байтовых элементов. После чего эти биты собираются в один байт и помещаются в младшие разряды регистра общего назначения. Подобная операция позволяет выполнять анализ знаков упакованных байт в MMX-регистре и организовывать ветвление программы.

ptomvmskb приемник, источник — команда для формирования байтового значения, биты которого представляют знаковые биты всех 8-ми байт, упакованных в MMX-регистре. *Источник* должен быть MMX-регистром. Результат из минимальных элементов каждой пары формируется в операнде *приемник*. *Приемник* является 32-битным регистром общего назначения (формируемый из знаковых разрядов байт будет являться младшим);

pmulhw приемник, источник — команда умножения упакованных беззнаковых слов с формированием в качестве результата старших слов произведения. *Источник* может быть MMX-регистром или 64-битной ячейкой памяти. Результат из старших слов произведения каждой пары формируется в операнде *приемник*. *Приемник* является MMX-регистром. Ранее мы уже рассматривали подобную команду — *pmulhw*, которая также работала со словами, но с учетом знака;

psadwb приемник, источник — команда вычисления суммарной разницы значений каждой пары беззнаковых байтов упакованных байтовых значений в *приемнике*

и *источнике*. Источник может быть MMX-регистром или 64-битной ячейкой памяти. Результат из старших слов произведения каждой пары формируется в операнде *приемник*. *Приемник* является MMX-регистром. Понимание алгоритма этой команды вызывает определенные трудности. Поэтому рассмотрим более детально:

- для каждой пары байтовых элементов операндов *приемник* и *источник* вычислить модуль их разности;
- сложить модули разности всех пар байтовых элементов и записать полученный результат в младшее слово *приемника*;
- старшие три слова операнда *приемник* обнуляются.

`pshufw приемник, источник, маска` — команда перераспределяет упакованные слова из операнда *источник* в операнд *приемник* в соответствии с *маской*, заданной третьим операндом. Источник может быть MMX-регистром или 64-битной ячейкой памяти. Результат перераспределения слов второго операнда помещается в операнд *приемник*, который является MMX-регистром;

Команда `pshufw` перераспределяет упакованные слова операнда *источник* в соответствии с *маской*. Содержимое *приемника* значения не имеет. *Маска* представляет собой байт, в котором:

- численное значение бит 0 и 1 определяют, какое из четырех слов источника необходимо разместить в 0 и 1 байте *приемника*;
- численное значение бит 2 и 3 определяют, какое из четырех слов *источника* необходимо разместить в 2 и 3 байте *приемника*;
- численное значение бит 4 и 5 определяют, какое из четырех слов *источника* необходимо разместить в 4 и 5 байте *приемника*;
- численное значение бит 6 и 7 определяют, какое из четырех слов *источника* необходимо разместить в 6 и 7 байте *приемника*.

Например, если операнд *источник* имеет значение 0012 950b 0054 0fd5 и задана *маска* 01101101b, то после применения команды `pshufw` в *приемнике* будет сформировано значение: 0054 950b 0012 0054.

XMM-расширение архитектуры микропроцессора Pentium

В 1999 году семейство микропроцессоров Pentium фирмы Intel пополнилось новой моделью — микропроцессором Pentium III (Katmai). Основу его архитектуры составляет ядро микропроцессора Pentium II, дополненное модулем SSE (Streaming SIMD Extensions — потоковое SIMD-расширение). Потоковое SIMD-расширение дополняет MMX-технологию средствами обработки данных с плавающей точкой. Выше мы условились называть этот тип MMX-расширения — *XMM-расширением*.

XMM-расширение является продолжением политики Intel на ввод в архитектуру микропроцессоров средств, повышающих эффективность вычислений на больших массивах однотипных данных. Программно-аппаратная модель XMM-расши-

рения имеет простую и гибкую структуру. В нее входит ряд новых регистров, новые команды и форматы данных. Программы, разработанные для ранних моделей микропроцессоров, будут также выполняться на компьютерах, оснащенных микропроцессором Pentium III, но им не будут доступны новшества, введенные XMM-расширением. Для этого программа должна быть изначально написана с учетом логики работы программно-аппаратных средств, входящих в XMM-расширение.

Модель XMM-расширения

XMM-расширение добавляет следующие новые элементы в архитектуру микропроцессора:

- восемь 128-битных регистров с плавающей точкой — `xmm0...xmm7` (XMM-регистры);
- формат данных (XMM-формат) размером 128 битов, представляющий собой совокупность из четырех упакованных 32-битных чисел с плавающей точкой в коротком формате — SPFP (Single Precision Floating Point);
- набор XMM-команд;
- 32-битный регистр управления/состояния.

Рассмотрим их подробнее.

На рис. 20.17 показаны восемь 128-битных XMM-регистров для хранения данных и один регистр управления/состояния. Каждый из этих регистров доступен соответствующими командами XMM-расширения.

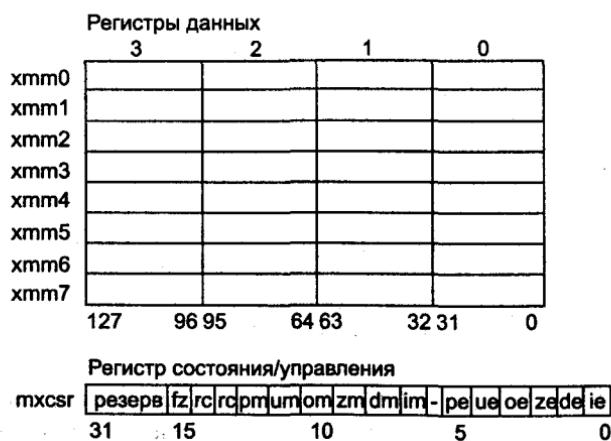


Рис. 20.17. XMM-регистры

XMM-регистры используются только для хранения данных. Не имеет смысла использовать их для хранения адресов памяти — для этого используются регистры общего назначения. При написании программы программист может смешивать команды обоих типов MMX-расширения, так как они используют физически разные наборы регистров. Вследствие этого для работы с ними не требуется использовать команду подобную `emms`.

Основной тип данных XMM-расширения — число с плавающей точкой в коротком формате. Операндом XMM-команды является XMM-регистр или 128-битная ячейка памяти, которые содержат четыре числа с плавающей точкой в коротком формате. Поэтому говорят, что числа упакованы и соответственно формат данных называется *упакованным форматом с плавающей точкой* (XMM-форматом). На уроке 19 нами рассматривались различные форматы данных сопроцессора, и данный формат был одним из них. Вспомним основные его характеристики:

- длина числа — 32 бита;
- длины полей в формате: мантиссы — 24 бита, порядка — 7 бит, знака — 1 бит;
- диапазон значений:
 - двоичное — $2^{-126} \dots 2^{127}$;
 - десятичное — $1,18 \times 10^{-38} \dots 1,70 \times 10^{38}$

Формат XMM-регистров или 128-битной ячейки памяти отражает структуру XMM-формата. Числа в *упакованном коротком формате с плавающей точкой* в пределах этих объектов нумеруются от 0 до 3 (рис. 20.18). Описание данных XMM-формата в программе на языке ассемблера производится по тем же принципам, которыми мы руководствовались при описании данных целочисленного MMX-расширения. В памяти данные в XMM-формате располагаются в соответствии с обычным для архитектуры Intel порядком — «младший байт по младшему адресу» (рис. 20.18).



Рис. 20.18. Схема расположения в памяти данных XMM-формата

Назначение регистра управления/состояния mxcsr аналогично соответствующим регистрам сопроцессора cwr и swr (урок 19). Так, большая часть бит этого регистра (табл. 20.8) используется для маскирования/размаскирования исключений, другая часть бит используется для установления способов округления, просмотра флагов состояния. Работа с этим регистром осуществляется с помощью команд ldmxcsr и fxrstor (загрузка содержимым из памяти) и stmxcsr и fxsave (сохранение содержимого mxcsr в памяти).

Таблица 20.8. Назначение бит регистра mxcsr

| Биты | Название | Назначение |
|-------|------------------------|--|
| 0...5 | ie, de, ze, oe, ue, pe | Фиксация исключений с плавающей точкой. Для очистки этих бит используется команда ldmxcsr. Возникающие исключения фиксируются для всего упакованного формата, и нет средств для определения того, какое конкретное число (из четырех) вызвало это исключение |

| Биты | Название | Назначение |
|---------|---|---|
| 7...12 | <code>im</code> , <code>dm</code> , <code>zm</code> , <code>om</code> , <code>um</code> , <code>pm</code> | Маскирование определенных типов исключений с плавающей точкой (урок 19). Начальное состояние этих битов единичное, что означает о маскировании всех исключений. Для установки этих битов используется команда <code>lfdmxcsr</code> . Маскирование исключений будет действовать для всего упакованного формата |
| 13...14 | <code>rc</code> (Rounding Control) | Задание режима округления. По умолчанию действует режим округления к ближайшему представимому значению числа в коротком формате с плавающей точкой. При желании можно установить следующие режимы округления: 00 – округление к ближайшему; 01 – округление в меньшую сторону; 10 – округление в большую сторону; 11 – округление к нулю с усечением, то есть отбрасыванием дробной части |
| 15 | <code>fz</code> (Flush-to-Zero) | Установка режима «сдвиг к нулю». Используется в ситуации переполнение следующим образом: <ul style="list-style-type: none"> • возвращается нулевое значение вместе с истинным знаком результата; • устанавливаются биты <code>UE</code> и <code>PE</code> в регистре <code>mxcsr</code>; По умолчанию бит установлен в 0. Наличие режима <code>fz</code> не соответствует стандарту IEE-754 на вычисления с плавающей точкой, так как этот стандарт требует при возникновении переполнения формировать денормализованный операнд. Этот режим введен из соображений эффективности. Ценой небольшой потери точности можно достигнуть более быстрой работы приложений, для которых ситуация переполнение обычна (так как не требуется вызова обработчика исключения). Размаскирование бита <code>UE</code> имеет приоритет перед режимом <code>f3flush-to-zero</code> |

Остальные биты регистра `mxcsr` (биты 31–16 и бит 6) зарезервированы и установлены в 0. При попытке записать ненулевое значение в эти биты будет возбуждено исключение общая защита.

Система команд

XMM-расширение дополнительно вводит в систему команд микропроцессора 70 новых машинных инструкций (рис. 20.19).

Дадим характеристику группам XMM-команд. Более подробную информацию по этим командам можно найти в Справочнике.

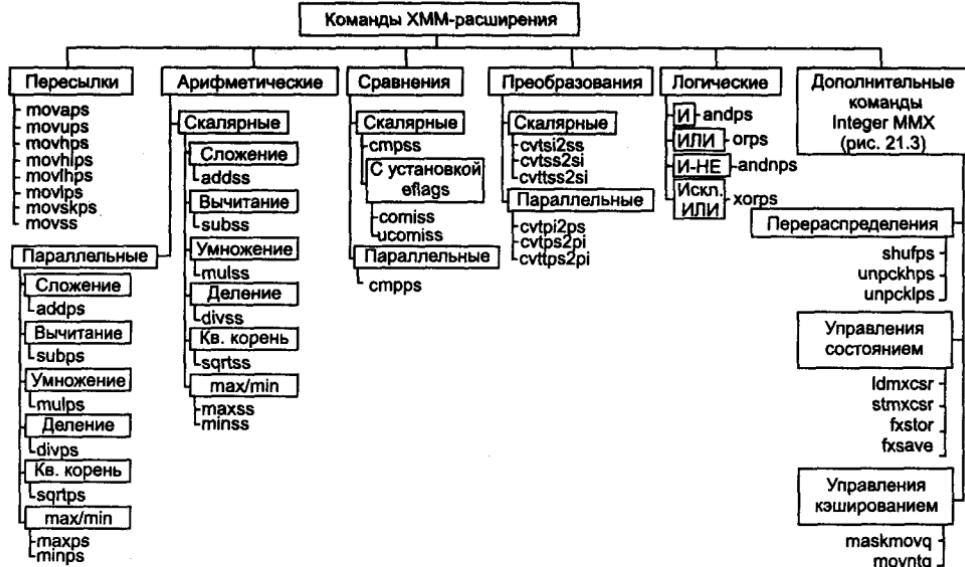


Рис. 20.19. Классификация команд ХММ-расширения

Команды перемещения данных

`movaps` *приемник, источник* — пересылка выровненных 128 битов из *источника* в *приемник*. Один из operandов, *источник* или *приемник* — ХММ-регистр. Другой operand должен быть ХММ-регистром или 128-разрядной ячейкой памяти. Адрес 128-разрядной ячейки памяти должен быть выровнен по 16-байтовой границе, иначе произойдет исключение общей защиты;

`movups` *приемник, источник* — пересылка не выровненных 128 битов из *источника* в *приемник*. Один из operandов, *источник* или *приемник* — ХММ-регистр. Другой operand должен быть ХММ-регистром или 128-разрядной ячейкой памяти. В отличие от команды `movaps`, данная команда не требует выравнивания по 16-байтовой границе адреса 28-разрядной ячейки памяти;

`movhps` *приемник, источник* — пересылка не выровненных 64 битов из *источника* в *приемник*. Один из operandов, *источник* или *приемник*, но не одновременно, должен быть ХММ-регистром. Другой operand — 64-разрядная ячейка памяти. Если пересылка производится из памяти, то 64 бита помещаются в старшие 64 бита ХММ-регистра, если пересылка производится из регистра, то пересылке подлежат старшие 64 бита ХММ-регистра. Младшие 64 бита ХММ-регистра не изменяются. Данная команда не требует выравнивания по 16-байтовой границе адреса 64-разрядной ячейки памяти;

`movhlps` *приемник, источник* — пересылка 64 битов из *источника* в *приемник*. Оба operandы должны быть ХММ-регистрами. В результате работы команды изменяются только содержимое младших 64 бит ХММ-регистра *приемник* в соответствии со схемой рис. 20.20;



Рис. 20.20. Схема работы команды movhlps

movlhp\$ приемник,источник — пересылка 64 бит из *источника* в *приемник*. Оба операнда должны быть XMM-регистрами. В результате работы команды изменяются только содержимое старших 64 бит XMM-регистра *приемника* в соответствии со схемой рис. 20.21;



Рис. 20.21. Схема работы команды movlhp\$

movlps приемник,источник — пересылка не выровненных 64 бит из *источника* в *приемник*. Один из operandов, *источник* или *приемник* — XMM-регистр. Другой operand должен быть 64-разрядной ячейкой памяти. Если пересылка производится из памяти, то 64 бита помещаются в младшие 64 бита XMM-регистра. Если пересылка производится из регистра, то пересылке подлежат младшие 64 бита регистра. Старшие 64 бита XMM-регистра не изменяются. Данная команда не требует выравнивания по 16-байтовой границе адреса 64-разрядной ячейки памяти;

movmskps приемник,источник — пересылка знакового бита каждого из четырех упакованных чисел с плавающей точкой *источника* в младшие четыре бита *приемника* (рис. 20.22). *Источник* должен быть XMM-регистром. *Приемник* должен быть 32-разрядным регистром общего назначения. В дальнейшем полученную четырехбитную величину можно использовать для организации условных переходов;

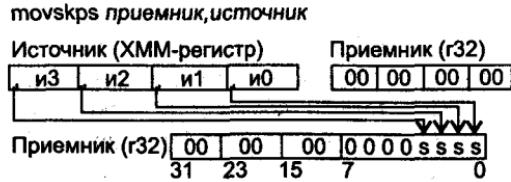


Рис. 20.22. Схема работы команды movmskps

movss приемник,источник — пересылка 32 младших бит из *источника* в *приемник*. Один из operandов или все operandы должны быть XMM-регистрами. Другой operand должен быть 32-разрядной ячейкой памяти. Если пересылка

производится из памяти, то 32 бита помещаются в младшие 32 бита XMM-регистра. Если пересылка производится из регистра, то пересылке подлежат младшие 32 бита регистра XMM. Остальные биты XMM-регистра не изменяются.

Арифметические команды

Набор арифметических команд XMM-расширения включает обычный набор команд для выполнения арифметических операций. Особенность этого набора в том, что он содержит два типа арифметических операций: скалярных и параллельных. Отличить эти команды можно по их мнемонике: скалярные команды имеют суффикс s, а параллельные — суффикс p. Команды *скалярных* арифметических операций обрабатывают только младшие 32-битные двойные слова упакованных операндов, остальные двойные слова в операции не участвуют и не изменяются. Команды *параллельных* арифметических операций обрабатывают одновременно четыре упакованных двойных слова. Заметьте также, что в наборе команд XMM-расширения присутствуют команды для операции деления, чего не было в целочисленном MMX-расширении.

Сложение и вычитание

addps (subps) *приемник, источник* — параллельное сложение (вычитание) элементов операндов *приемник* и *источник*. Операнды должны иметь XMM-формат. Результат операции помещается в *приемник* — XMM-регистр. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти;

addss (subss) *приемник, источник* — скалярное сложение (вычитание) операндов *приемник* и *источник*. Младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате. Результат помещается в operand *приемник*. *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 32-битной ячейкой памяти. Содержимое старших бит операндов не имеет значения и не изменяется.

Умножение и деление

mulp (divp) *приемник, источник* — параллельное умножение (деление) операнда *приемник* на operand *источник*. Операнды должны иметь XMM-формат. Результат операции помещается в *приемник* — XMM-регистр. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти;

mulss (divss) *приемник, источник* — скалярное умножение (деление) операнда *приемник* на operand *источник*. Младшие двойные слова операндов должны быть числами с плавающей точкой в коротком формате. Результат помещается в operand *приемник*. *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 32-битной ячейкой памяти. Содержимое старших бит операндов не имеет значения и не изменяется.

Извлечение квадратного корня

sqrtps *приемник, источник* — параллельное извлечение квадратного корня из упакованных чисел с плавающей точкой operand *источник* в XMM-формате. Результат помещается в operand *приемник* — XMM-регистр. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти;

sqrtss *приемник, источник* — скалярное извлечение квадратного корня из упакованного числа с плавающей точкой операнда *источник* в XMM-формате. Результат помещается в operand *приемник*. *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 32-битной ячейкой памяти. Если *источник* - XMM-регистр, то в операции извлечения участвует только число с плавающей точкой в его младшем двойном слове. Содержимое старших бит operandов не имеет значения и не изменяется.

Извлечение максимальных/минимальных значений operandов

maxps (*minps*) *приемник, источник* — параллельное извлечение максимальных (минимальных) значений из каждой пары упакованных чисел с плавающей точкой operandов *источник* и *приемник* в XMM-формате. Результат формируется из максимальных (минимальных) значений каждой пары и помещается в operand *приемник* — XMM-регистр. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти, и его содержимое не изменяется;

maxss (*minss*) *приемник, источник* — скалярное извлечение максимального (минимального) значения из младшей пары упакованных чисел с плавающей точкой operandов *источник* и *приемник* в XMM-формате. Результат формируется из максимального (минимального) значений каждой пары и помещается в operand *приемник* — XMM-регистр. *Источник* может быть XMM-регистром или 32-битной ячейкой памяти, и его содержимое не изменяется. Если *источник* — XMM-регистр, то в операции сравнения и извлечения участвует только число с плавающей точкой в его младшем двойном слове.

Команды сравнения

Команды сравнения также делятся на два типа: скалярные и параллельные. Команды сравнения или их комбинации поддерживают полный набор условий сравнения. Обратите внимание на формат команд — он несколько необычен, так как условие сравнения задано непосредственным operandом.

cmpps *приемник, источник, условие* — параллельное сравнение значений каждой пары упакованных чисел с плавающей точкой operandов *источник* и *приемник* в XMM-формате. Условие, по которому производится сравнение, определяется третьим operandом, представляющим собой непосредственное значение. Результат сравнения формируется в operandе *приемник* — XMM-регистр. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение *приемника*, которое замещается 32-битными значениями 00000000h или 0fffffffh в зависимости от результата сравнения (00000000h — если условие не выполняется для данной пары значений упакованных чисел operandов *источник* и *приемник*, и 0fffffffh, если условие выполняется);

cmpss *приемник, источник, условие* — скалярное сравнение младших пар чисел с плавающей точкой в коротком формате operandов *приемник* и *источник*. Условие, по которому производится сравнение, определяется третьим operandом, представляющим собой непосредственное значение. Результат сравнения формируется в operandе *приемник* — XMM-регистр. *Источник* может быть XMM-регистром или 32-битной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение *приемника*, младшее двойное слово которого замещается 32-битным значением 00000000h или 0fffffffh в зависимости от

результата сравнения (00000000h – если условие не выполняется для младшей пары чисел с плавающей точкой в коротком формате из операндов *источник* и *приемник*, и 0fffffffh, если условие выполняется);

Предыдущие две команды сравнивали операнды, изменяя при этом содержимое операнда *приемник*. Следующие две команды позволяют выполнить операцию сравнения без изменения его содержимого, но с одним ограничением — эти команды являются скалярными. По результатам их работы устанавливаются флаги в регистре eflags.

comiss приемник, источник — скалярное сравнение младших пар чисел с плавающей точкой в коротком формате операндов *приемник* и *источник*. Условие, по которому производится сравнение, определяется третьим операндом, представляющим собой непосредственное значение. Результат сравнения формируется установкой флагов в регистре eflags (устанавливаются флаги zf, pf, cf и очищаются флаги of, sf, af). *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 32-битной ячейкой памяти. Содержимое *приемника* и *источника* не изменяется;

icomiss приемник, источник — неупорядоченное скалярное сравнение младших пар чисел с плавающей точкой в коротком формате операндов *приемник* и *источник*. Данная команда отличается от команды *comiss* тем, что позволяет обнаружить ситуацию, когда младшие 32 бита операнда *приемника* являются тихим или сигнальным не числом. Результат сравнения формируется установкой флагов в регистре eflags (устанавливаются флаги zf, pf, cf и очищаются флаги of, sf, af). *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 32-битной ячейкой памяти. Содержимое *приемника* и *источника* не изменяются.

Команды преобразования

Данная группа команд позволяет выполнить взаимное преобразование значений трех форматов: XMM-формата, MMX-формата и обычного целочисленного двоичного формата представления числа в одном из регистров общего назначения. Набор команд также делится на два класса: скалярные и параллельные.

cvtpr2ps приемник, источник — параллельное преобразование двух 32-битных целочисленных значений в операнде *источник* — MMX-регистр целочисленного MMX-расширения или 64-битная ячейка памяти, содержащие два 32-битных целочисленных значения. Результат преобразования — в виде двух 32-битных чисел с плавающей точкой в коротком формате. Эти значения размещаются в двух младших двойных словах XMM-регистра, представляющего собой операнд *приемник*. Старшие два двойных слова в операнде *приемник* не изменяются. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра mxcsr;

cvtps2ri приемник, источник — параллельное преобразование двух 32-битных чисел с плавающей точкой в коротком формате, содержащихся в двух младших двойных словах операнда *источник*, в два 32-битных целых значения. Операнд *источник* представляет собой XMM-регистр или 64-битную ячейку памяти. Операнд *приемник* представляет собой адрес 64-битной ячейки памяти. Данная

команда по своему действию является обратной команде `cvtpt2ps`. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра `mxcsr`.

В системе команд XMM-расширения существует другой вариант последней команды — `cvtpts2pi`, которая перед преобразованием отбрасывает дробную часть операнда *источник*.

`cvttsi2ss приемник, источник` — скалярное преобразование одного 32-битного целочисленного значения из операнда *источник* в одно 32-битное число с плавающей точкой в коротком формате в операнде *приемник*. Операнд *источник* представляет собой один из 32-битных регистров общего назначения или 32-битную ячейку памяти. Результат преобразования в виде одного 32-битного числа с плавающей точкой в коротком формате помещается в младшее двойное слово 128-битного операнда *приемник*, представляющий собой XMM-регистр. Старшие три двойных слова в операнде *приемник* не изменяются. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра `mxcsr`;

`cvtss2si приемник, источник` — скалярное преобразование одного 32-битного числа с плавающей точкой в коротком формате из операнда *источник* в одно 32-битное целое число в операнде *приемник*. Операнд *источник* представляет XMM-регистр или 32-битную ячейку памяти. Результат преобразования в виде одного 32-битного целочисленного значения помещается в operand *приемник*, представляющий собой 32-битный регистр общего назначения. Старшие три двойных слова в операнде *приемник* не изменяются. Если не удается получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра `mxcsr`.

В системе команд существует другой вариант последней команды — `cvttss2si`, которая перед преобразованием отбрасывает дробную часть операнда *источник*.

Логические команды

Все команды данной группы являются параллельными и реализуют следующие логические операции над парами битов в обоих операндах: «И», «И-НЕ», «ИЛИ», «исключающее ИЛИ».

`andps приемник, источник` — параллельное выполнение логического умножения (операция логического «И») над парами битов упакованных чисел с плавающей точкой операндов *источник* и *приемник* в XMM-формате. Результат операции логического умножения формируется в операнде *приемник* — XMM-регистре. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение *приемника*;

`andnp приемник, источник` — параллельное выполнение логической операции «И-НЕ» над парами битов упакованных чисел с плавающей точкой операндов *источник* и *приемник* в XMM-формате. Результат выполнения логической операции «И-НЕ» формируется в операнде *приемник*. *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение *приемника*;

`orps приемник, источник` — параллельное выполнение логического сложения (операция логического «ИЛИ») над парами бит упакованных чисел с плавающей точкой операндов *источник* и *приемник* в XMM-формате. Результат операции логического сложения формируется в операнде *приемник*. *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение *приемника*;

`xorps приемник, источник` — параллельное выполнение логической операции «исключающего ИЛИ») над парами бит упакованных чисел с плавающей точкой операндов *источник* и *приемник* в XMM-формате. Результат логической операции «исключающего ИЛИ» формируется в операнде *приемник*. *Приемник* должен быть XMM-регистром. *Источник* может быть XMM-регистром или 128-битной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение *приемника*.

Команды управления состоянием

`ldmxcsr источник` — команда загрузки регистра *mxcsr* (регистра состояния и управления XMM-расширения) из памяти. *Источник* должен быть 32-битной ячейкой памяти;

`stmxcsr приемник` — команда сохранения содержимого регистра *mxcsr* (регистра состояния и управления XMM-расширения) в памяти. *Приемник* должен быть 32-битной ячейкой памяти;

`fxrstor источник` — команда загрузки состояния сопроцессора (целочисленного MMX-расширения) и XMM-расширения в области памяти размером 512 байт;

`fxsave приемник` — команда сохранения состояния сопроцессора (целочисленного MMX-расширения) и XMM-расширения в области памяти размером 512 байт. Подобно команде `fsave`, команда `fxsave` не инициализирует сопроцессор.

Команды перераспределения

`shufps приемник, источник, маска` — команда упакованной перестановки двойных слов из операндов *источник* и *приемник* в operand *приемник* в соответствии с *маской*, заданной третьим операндом. *Источник* может быть MMX-регистром или 128-битной ячейкой памяти. Результат перераспределения двойных слов обоих операндов помещается в operand *приемник*, который является MMX-регистром. Третий operand является непосредственным operandом. Принцип формирования результата следующий. Младшие два двойных слова результата формируют два из четырех двойных слов *приемника*. Старшие два двойных слова результата формируют два из четырех двойных слов *источника*. Какие именно двойные слова будут включены в качестве двойных слов результата, определяют биты *маски*. Значение первых двух битов *маски* определяют номер двойного слова *источника*, которое будет включено в результат в качестве первого двойного слова. Значение следующей пары битов (2 и 3 биты *маски*) определяют номер двойного слова *источника*, которое будет включено в результат в качестве второго двойного слова. Значение следующей пары битов *маски* (4 и 5) определяют номер двойного слова *приемника*, которое будет включено в

результат в качестве третьего двойного слова. Последняя пара битов *маски* (6 и 7) определяют номер двойного слова *приемника*, которое будет включено в результат в качестве четвертого двойного слова.

`unprckhps приемник,источник` — команда упакованного перемещения с чередованием старших двух двойных слов из операндов *источник* и *приемник* в операнд *приемник*. *Источник* может быть MMX-регистром или 128-битной ячейкой памяти. Результат перемещения двойных слов обоих операндов формируется в операнде *приемник*, который является MMX-регистром. Принцип формирования результата показан на рис. 20.23.



Рис. 20.23. Схема работы команды `unprckhps`

`unprcklps приемник,источник` — команда упакованного перемещения с чередованием младших двух двойных слов из операндов *источник* и *приемник* в операнд *приемник*. *Источник* может быть MMX-регистром или 128-битной ячейкой памяти. Результат перемещения двойных слов обоих операндов формируется в операнде *приемник*, который является MMX-регистром. Принцип формирования результата показан на рис. 20.24.

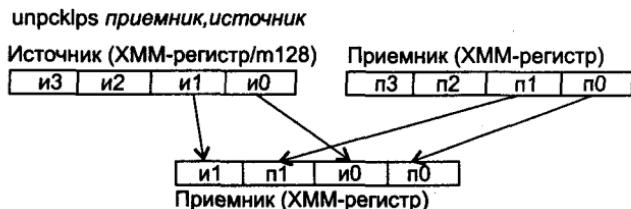


Рис. 20.24. Схема работы команды `unprcklps`

Команды управления кэшированием

Команды ХММ-расширения, рассмотренные в этом разделе, позволяют программисту управлять кэшированием данных. До этой модели микропроцессора в системе команд микропроцессора Intel было несколько команд, которые влияли на работу кэша, но их работа не отличалась гибкостью и с их помощью тяжело было учесть особенности конкретной задачи. Потребность оптимизировать работу кэша возникла из-за большого объема данных, которыми манипулирует большинство мультимедийных приложений. Чтобы предотвратить загрузку в кэш данных, которые могут впоследствии не понадобиться вообще, были введены специальные команды. Более того, с помощью новых команд можно организовать предварительную загрузку данных, то есть до того, как они реально будут востребованы. Понятно, что это поднимет скорость работы приложений. Рассмотрим эти команды.

Вначале будут рассмотрены три команды, которые обеспечивают программное управление записью данных в память из регистров целочисленного или потокового MMX-расширений, обеспечивая при этом минимизацию загрязнения кэш-памяти.

maskmovq *источник, маска* — команда выборочного сохранения в памяти байтов упакованного целого числа из MMX-регистра. *Источник* является MMX-регистром целочисленного MMX-расширения. Местоположение *приемника* в памяти задано неявно содержимым пары *ds:di/edi*. *Маска* определяет, какие байты будут сохранены в памяти следующим образом. Значение для данной команды имеет знаковый разряд каждого упакованного байта *маски*. Если он равен 1, то значение соответствующего байта из MMX-регистра *источник* переносится в соответствующий байт памяти. Если же он равен 0, то содержимое соответствующего байта из MMX-регистра *источника* в память не переносится. Наглядно этот процесс показан на рис. 20.25. Особенность этой команды (и собственно, причина, по которой она включена в данную группу команд) то, что при записи информации из регистров в память не производится кэширования этой информации и таким образом загрязнения кэш-памяти первого или второго уровня.



Рис. 20.25. Схема работы команды **maskmovq**

movntq *приемник, источник* — команда сохранения в памяти упакованных целых чисел в целочисленном MMX-формате. *Источник* является MMX-регистром целочисленного MMX-расширения. *Приемником* является 64-битная ячейка памяти. Запись в память производится, минуя кэш-память.

movnpes *приемник, источник* — команда сохранения в памяти упакованных чисел с плавающей точкой в ХММ-формате. *Источник* является XMM-регистром. *Приемником* является 128-битная ячейка памяти, адрес которой должен быть выравнен на границу параграфа (кратную 16). Запись в память производится, минуя кэш-память.

Таким образом, приведенные выше три команды отличаются от обычных команд пересылки политикой работы с кэш-памятью.

Подведем некоторые итоги:

- ✓ В основе MMX-технологии лежит принцип SIMD – Single Instruction Multiple Data. Согласно этому принципу, обработка нескольких одинаковых по размеру элементов данных производится параллельно одной командой.

- MMX-расширение поддерживает несколько новых типов данных, представляющих собой совокупность упакованных байт, слов, двойных слов и одно учетверенное слово. Эти типы данных имеют общий размер 64 бита, что согласуется с размером шины данных микропроцессора Pentium.
- Аппаратно MMX-расширение реализовано в виде восьми регистров и дополнительного набора из 57 машинных команд. MMX-регистры имеют размер 64 бита и физически соответствуют регистрам сопроцессора.
- Важное значение имеет особенность формирования результатов работы MMX-команд при возникновении ситуации переполнения. Команды целочисленного устройства реагируют на нее в соответствии с принципом циклического переполнения. Среди MMX-команд также есть команды, работающие в соответствии с этим принципом, но большинство этих команд используют принцип знакового и беззнакового насыщения.
- Далеко не все трансляторы ассемблера поддерживают мнемоническое обозначение MMX-команд, поэтому их необходимо уметь моделировать. При моделировании важно учитывать, в какой среде — 16- или 32-разрядной, будет работать разрабатываемое приложение.
- Для отладки программ, использующих MMX-расширение, можно использовать обычные отладчики, главное требование к которым заключается в том, чтобы они поддерживали работу с сопроцессором. Если соблюдать (и что немаловажно, понимать) определенные ограничения при работе с отладчиком, то особых проблем, как правило, не возникает. Что касается отладчика Turbo Debugger, то необходимо, чтобы во время работы с MMX-командами активным было окно CPU отладчика.
- Если вы разрабатываете приложение с использованием MMX-расширения для коммерческого (или некоммерческого) распространения, то вам необходимо предусматривать то, что возможно компьютер, на котором ваше приложение будет работать, и не поддерживает MMX-команд. Для этого вы должны предварительно, используя команду `cputid`, установить этот факт, после чего передать управление на соответствующую ветвь программы: содержащую MMX-команды или не содержащую этих команд. Главное, чтобы эти две ветви реализовывали одинаковые функции.
- В дополнение к целочисленному MMX-расширению, фирма Intel разработала MMX-расширение с плавающей точкой (XMM-расширение). Соответствующий аппаратный модуль SSE и система команд к нему были включены в архитектуру микропроцессора Pentium III.
- Аппаратная часть SSE-модуля представляет собой восемь 128-битных регистров и один регистр управления.
- Новое расширение оперирует с данными в XMM-формате, представляющими собой четыре упакованных вещественных числа в коротком формате с плавающей точкой.

- Система команд XMM-расширения обладает большей гибкостью по сравнению с MMX-расширением. В нее введены, кроме традиционных команд работы с данными, средства для различных перемещений упакованных элементов.
- В системе команд XMM-расширения есть команда деления, которой не было в MMX-расширении. Также XMM-расширение содержит команды взаимного конвертирования данных для трех форматов — XMM-формата, MMX-формата и обычного целочисленного двоичного формата.
- В систему команд Pentium III введены команды, которые позволяют управлять кэшированием данных. Как правило, массивы мультимедийных данных большие и для работы с ними просто нет необходимости использовать механизм кэширования. Команды группы управления кэшированием позволяют производить работу с данными в памяти в обход механизма кэширования.
- В систему команд Pentium III также включены дополнительные команды для целочисленного MMX-расширения.

Вместо заключения...

Ну вот и все! Прощаясь с читателем, позволю себе немного порассуждать.

В этом учебнике я попытался рассмотреть язык ассемблера как обычный язык программирования. При этом мне хотелось показать, что этот язык ничем не хуже, а в чем-то даже лучше привычных читателю языков высокого уровня. Осознание этого особенно важно для начинающих программистов, которые зачастую с первых шагов в своей карьере попадают в «теплые, нежные, заботливые... (далее придумайте сами)» объятия различных интегрированных оболочек, сред, студий... Опасно это тем, что такой молодой неопытный и наивный программист постепенно всецело поддается нежному шепоту этих интегрированных продуктов и засыпает. Спать он может долго, и, возможно, при этом бездлаже будет сниться сон, в котором:

- не нужно думать об оптимизации программы, ведь объема памяти и запаса быстродействия у современного процессора хватит на мгновенное псевдопараллельное выполнение и десятка даже самых бездарных программ;
- не нужно думать о работе с «железом», ведь современная ОС имеет столько драйверов устройств, что вряд ли встретится ситуация, когда к компьютеру потребуется подключить что-то нестандартное;
- не нужно думать о защите своих программ — от вирусов нас защитит программа-антивирус;
- Боже упаси лезть в чужой исполняемый код, даже если лопаешься от любопытства;
- и, наконец, нет необходимости что-то знать и о самом компьютере — стоит себе ящик, работает, когда его включишь, — чего еще нужно?

Обычно сон прерывается в тот момент, когда человек осознает, что сидит перед совершенно конкретным компьютером, функционирующим на основе определенных программно-аппаратных средств, со всеми их изъянами и достоинствами, и которые иногда требуют прямого вмешательства человека в свою работу. В этом случае можно, конечно, обратиться к специалисту, но какой же ты тогда профессионал?.. Судить обо всем этом, конечно, читателю. Может быть, я в чем-то и неправ. Ведь компьютер уже давно стал повседневным рабочим инструментом людей множества профессий, которым действительно все сказанное выше «до лампочки». Но если книга у вас в руках, то с большой долей вероятности можно предположить, что я не одинок в своих мыслях, и вы человек, страдающий профессиональной бессонницей. Вас трудно ввести в заблуждение обещаниями легкого достижения поставленных вами целей, и вы способны самостоятельно выбрать кратчайший путь для достижения вершины своей карьеры.

Успехов вам на этом пути!

Краткое содержание

| | |
|--|-----|
| Предисловие | 9 |
| Урок 1. Общие сведения об ЭВМ | 14 |
| Урок 2. Архитектура персонального компьютера | 26 |
| Урок 3. Простая программа на ассемблере | 57 |
| Урок 4. Жизненный цикл программы на ассемблере | 66 |
| Урок 5. Структура программы на ассемблере | 86 |
| Урок 6. Система команд микропроцессора | 111 |
| Урок 7. Команды обмена данными | 133 |
| Урок 8. Арифметические команды | 153 |
| Урок 9. Логические команды | 184 |
| Урок 10. Команды передачи управления | 202 |
| Урок 11. Цепочечные команды | 229 |
| Урок 12. Сложные структуры данных | 250 |
| Урок 13. Макросредства языка ассемблера | 278 |
| Урок 14. Модульное программирование | 310 |
| Урок 15. Прерывания | 353 |
| Урок 16. Защищенный режим работы микропроцессора | 380 |
| Урок 17. Обработка прерываний в защищенном режиме | 404 |
| Урок 18. Создание Windows-приложений на ассемблере | 419 |
| Урок 19. Архитектура и программирование сопроцессора | 500 |
| Урок 20. MMX-технология микропроцессоров Intel | 564 |
| Вместо заключения... | 623 |

Содержание

| | |
|---|-----|
| Предисловие | 9 |
| Урок 1. Общие сведения об ЭВМ | 14 |
| Урок 2. Архитектура персонального компьютера | 26 |
| Архитектура ЭВМ | 27 |
| Набор регистров | 38 |
| Организация памяти | 46 |
| Типы данных | 51 |
| Формат команд | 54 |
| Обработка прерываний | 55 |
| Урок 3. Простая программа на ассемблере | 57 |
| Урок 4. Жизненный цикл программы на ассемблере | 66 |
| Трансляция программы | 71 |
| Компоновка программы | 77 |
| Отладка программы | 79 |
| Утилита MAKE | 83 |
| Урок 5. Структура программы на ассемблере | 86 |
| Синтаксис ассемблера | 88 |
| Директивы сегментации | 98 |
| Простые типы данных ассемблера | 105 |
| Урок 6. Система команд микропроцессора | 111 |
| Системы счисления | 112 |
| Двоичная система счисления | 113 |
| Шестнадцатеричная система счисления | 114 |
| Десятичная система счисления | 116 |
| Перевод чисел из одной системы счисления в другую | 116 |
| Перевод в десятичную систему счисления | 116 |
| Перевод в двоичную систему счисления | 116 |
| Перевод в шестнадцатеричную систему счисления | 117 |
| Перевод дробных чисел | 118 |
| Числа со знаком | 121 |
| Структура машинной команды | 123 |
| Способы задания операндов команды | 126 |
| Функциональная классификация машинных команд | 131 |

| | |
|---|------------|
| Урок 7. Команды обмена данными | 133 |
| Пересылка данных | 135 |
| Ввод-вывод в порт | 137 |
| Работа с адресами и указателями | 143 |
| Преобразование данных | 145 |
| Работа со стеком | 147 |
| Урок 8. Арифметические команды | 153 |
| Общий обзор | 154 |
| Целые двоичные числа | 155 |
| Десятичные числа | 157 |
| Арифметические операции над целыми двоичными числами | 159 |
| Сложение двоичных чисел без знака | 159 |
| Сложение двоичных чисел со знаком | 160 |
| Вычитание двоичных чисел без знака | 162 |
| Вычитание двоичных чисел со знаком | 164 |
| Вычитание и сложение операндов большой размерности | 165 |
| Умножение двоичных чисел без знака | 166 |
| Умножение двоичных чисел со знаком | 168 |
| Деление двоичных чисел без знака | 168 |
| Деление двоичных чисел со знаком | 170 |
| Вспомогательные команды для целочисленных операций | 170 |
| Команды преобразования типов | 170 |
| Другие полезные команды | 172 |
| Арифметические операции над двоично-десятичными числами | 173 |
| Неупакованные BCD-числа | 174 |
| Упакованные BCD-числа | 180 |
| Урок 9. Логические команды | 184 |
| Логические данные | 186 |
| Логические команды | 187 |
| Команды сдвига | 191 |
| Линейный сдвиг | 192 |
| Циклический сдвиг | 194 |
| Дополнительные команды сдвига | 196 |
| Примеры работы с битовыми строками | 198 |
| Рассогласование битовых строк | 198 |
| Вставка битовых строк | 199 |
| Извлечение битовых строк | 200 |
| Пересылка битов | 200 |
| Урок 10. Команды передачи управления | 202 |
| Безусловные переходы | 208 |
| Команда безусловного перехода jmp | 208 |
| Процедуры | 212 |
| Условные переходы | 217 |
| Команда сравнения cmp | 218 |
| Команды условного перехода и флаги | 220 |
| Команды условного перехода и регистр esch/cx | 222 |
| Организация циклов | 222 |
| Урок 11. Цепочечные команды | 229 |
| Пересылка цепочек | 233 |
| Команда movs | 234 |
| Команды пересылки байтов, слов и двойных слов | 235 |

| | |
|---|------------|
| Сравнение цепочек | 236 |
| Команда cmps | 236 |
| Команды сравнения байтов, слов и двойных слов | 239 |
| Сканирование цепочек | 239 |
| Команда scas | 240 |
| Сканирование строки байтов, слов, двойных слов | 242 |
| Загрузка элемента цепочки в аккумулятор | 242 |
| Команда lods | 242 |
| Загрузка в регистр al/ax/eax байтов, слов, двойных слов | 244 |
| Перенос элемента из аккумулятора в цепочку | 244 |
| Команда stos | 244 |
| Сохранение в цепочке байта, слова, двойного слова из регистра al/ax/eax | 246 |
| Ввод элемента цепочки из порта ввода-вывода | 247 |
| Выход элемента цепочки в порт ввода-вывода | 247 |
| Урок 12. Сложные структуры данных | 250 |
| Массивы | 252 |
| Описание и инициализация массива в программе | 252 |
| Доступ к элементам массива | 254 |
| Типовые операции с массивами | 261 |
| Структуры | 264 |
| Описание шаблона структуры | 265 |
| Определение данных с типом структуры | 265 |
| Методы работы со структурой | 266 |
| Объединения | 268 |
| Записи | 270 |
| Описание записи | 271 |
| Определение экземпляра записи | 271 |
| Работа с записями | 273 |
| Дополнительные возможности обработки | 275 |
| Урок 13. Макросредства языка ассемблера | 278 |
| Псевдооператоры equ и = | 280 |
| Макрокоманды | 283 |
| Макродирективы | 291 |
| Директивы WHILE и REPT | 292 |
| Директива IRP | 293 |
| Директива IRPC | 294 |
| Директивы условной компиляции | 294 |
| Директивы компиляции по условию | 295 |
| Директивы генерации ошибок | 302 |
| Константные выражения в условных директивах | 305 |
| Дополнительное управление трансляцией | 307 |
| Урок 14. Модульное программирование | 310 |
| Технологии программирования | 311 |
| Структурное программирование | 312 |
| Концепция модульного программирования | 313 |
| Процедуры в языке ассемблера | 315 |
| Организация интерфейса с процедурой | 317 |
| Связь ассемблера с языками высокого уровня | 330 |
| Связь Pascal—ассемблер | 333 |
| Команды enter и leave | 343 |
| Связь C—ассемблер | 346 |

| | |
|---|-----|
| Урок 15. Прерывания | 353 |
| Контроллер прерываний | 357 |
| Программирование контроллера прерываний i8259A | 362 |
| ICW1 — определить особенности последовательности приказов | 363 |
| ICW2 — определение базового адреса | 363 |
| ICW3 — связь контроллеров | 364 |
| ICW4 — дополнительные особенности обработки прерываний | 366 |
| OCW1 — управление регистром масок IMR | 366 |
| OCW2 — управление приоритетом | 366 |
| OCW3 — общее управление контроллером | 367 |
| Каскадирование микросхем i8259A | 368 |
| Реальный режим работы микропроцессора | 369 |
| Обработка прерываний в реальном режиме | 371 |
| Урок 16. Защищенный режим работы микропроцессора | 380 |
| Системные регистры микропроцессора | 382 |
| Регистры управления | 383 |
| Регистры системных адресов | 384 |
| Регистры отладки | 385 |
| Структуры данных защищенного режима | 386 |
| Пример программы защищенного режима | 391 |
| Подготовка таблиц глобальных дескрипторов GDT | 392 |
| Запрет обработки аппаратных прерываний | 397 |
| Переключение микропроцессора в защищенный режим | 398 |
| Работа в защищенном режиме | 399 |
| Переключение микропроцессора в реальный режим | 401 |
| Разрешение прерываний | 402 |
| Стандартное для MS-DOS завершение работы программы | 402 |
| Урок 17. Обработка прерываний в защищенном режиме | 404 |
| Шлюз ловушки | 411 |
| Шлюз прерывания | 412 |
| Шлюз задачи | 413 |
| Инициализация таблицы IDT | 415 |
| Обработчики прерываний | 416 |
| Программирование контроллера прерываний 8259A | 416 |
| Загрузка регистра IDTR | 417 |
| Урок 18. Создание Windows-приложений на ассемблере | 419 |
| Каркасное Windows-приложение на C/C++ | 423 |
| Каркасное Windows-приложение на ассемблере | 435 |
| Стартовый код (строки 54–73) | 445 |
| Главная функция (строки 74–162) | 447 |
| Обработка сообщений в оконной функции | 456 |
| Средства TASM для разработки Windows-приложений | 459 |
| Углубленное программирование на ассемблере для Win32 | 462 |
| Ресурсы Windows-приложений на языке ассемблера | 463 |
| Меню в Windows-приложениях | 464 |
| Перерисовка изображения | 472 |
| Окна диалога в Windows-приложениях | 480 |
| Урок 19. Архитектура и программирование сопроцессора | 500 |
| Архитектура сопроцессора | 502 |
| Регистр состояния swr | 507 |

| | |
|--|------------|
| Регистр управления <i>cwr</i> | 508 |
| Регистр тегов <i>tvr</i> | 509 |
| Форматы данных | 509 |
| Двоичные целые числа | 510 |
| Упакованные целые десятичные (BCD) числа | 511 |
| Вещественные числа | 512 |
| Специальные числовые значения | 516 |
| Система команд сопроцессора | 520 |
| Команды передачи данных | 522 |
| Команды загрузки констант | 525 |
| Команды сравнения данных | 525 |
| Арифметические команды | 528 |
| Команды трансцендентных функций | 537 |
| Команды управления сопроцессором | 549 |
| Исключения сопроцессора и их обработка | 554 |
| Исключение недействительная операция | 555 |
| Деление на ноль | 556 |
| Денормализация операнда | 556 |
| Переполнение и антипереполнение | 556 |
| Неточный результат | 556 |
| Немаскируемая обработка исключений | 557 |
| Использование отладчика | 559 |
| Общие рекомендации по программированию сопроцессора | 561 |
| Урок 20. MMX-технология микропроцессоров Intel | 564 |
| MMX-расширение архитектуры микропроцессора Pentium | 566 |
| Модель целочисленного MMX-расширения | 566 |
| Особенности команд MMX-расширение | 568 |
| Система команд | 574 |
| Отладка программ | 578 |
| Пример применения MMX-технологии | 593 |
| Дополнительные целочисленные MMX-команды (Pentium III) | 606 |
| ХММ-расширение архитектуры микропроцессора Pentium | 608 |
| Модель ХММ-расширения | 609 |
| Система команд | 611 |
| Вместо заключения... | 623 |