КЛАССИКА COMPUTER SCIENCE

АРХИТЕКТУРА КОМПЬЮТЕРА

6-Е ИЗДАНИЕ

Э. ТАНЕНБАУМ Т. ОСТИН

PH PTR

ПИТЕР®

STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, The Netherlands

TODD AUSTIN
University of Michigan
Ann Arbor, Michigan, United States

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

КЛАССИКА COMPUTER SCIENCE

Э. ТАНЕНБАУМ, Т. ОСТИН

АРХИТЕКТУРА КОМПЬЮТЕРА

6-Е ИЗДАНИЕ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Екатеринбург · Самара · Минск

2021

ББК 32.973.23-02
УДК 004.3
Т18

Таненбаум Э., Остин Т.

Т18 Архитектура компьютера. 6-е изд. — СПб.: Питер, 2021. — 816 с.: ил. — (Серия «Классика computer science»).

ISBN 978-5-4461-1103-9

Книга Эндрю Таненбаума, всемирно известного специалиста в области информационных технологий, писателя и преподавателя, выходящая уже в шестом издании, посвящена структурной организации компьютера. В ее основе лежит идея иерархической структуры, в которой каждый уровень выполняет вполне определенную функцию. В рамках этого нетрадиционного подхода подробно описываются цифровой логический уровень, уровень архитектуры команд, уровень операционной системы и уровень языка ассемблера. В шестое издание внесены многочисленные изменения, которые приводят книгу в соответствие со стремительным развитием компьютерной отрасли. В частности, была обновлена информация о машинах, представленных в качестве примеров: Intel Core i7, Texas Instrument OMAP4430 и Atmel ATmega168.

Книга рассчитана на широкий круг читателей: как на студентов, изучающих компьютерные технологии, так и на тех, кто самостоятельно знакомится с архитектурой компьютера.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.23-02
УДК 004.3

ISBN 978-0132916523 англ.
ISBN 978-5-4461-1103-9

Краткое оглавление

Оглавление

Э. Таненбаум (AST): Сюзанне, Барбаре, Марвину, Арону
и Н. Т. Остин (TA): Роберте, которая предоставила мне место
(и время) для завершения этого проекта

Предисловие

В основе первых пяти изданий книги лежит идея о том, что компьютер можно рассматривать как иерархию уровней, каждый из которых выполняет какую-либо определенную функцию. Это фундаментальное утверждение сейчас столь же правомерно, как в момент выхода в свет первого издания, поэтому я по-прежнему беру его за основу, на этот раз уже в шестом издании. Как и в первых пяти, в этом подробно описываются цифровой логический уровень, а также уровни микроархитектуры, архитектуры набора команд, операционной системы и ассемблера.

В целом структура книги осталась прежней, но в шестое издание внесены многочисленные изменения, которые приводят ее в соответствие со стремительным развитием компьютерной отрасли. В частности, были обновлены машины, представленные в качестве примеров. В этом издании рассматриваются Intel Core i7, Texas Instrument OMAP4430 и Atmel ATmega168. Core i7 — популярный процессор, используемый в портативных и настольных компьютерах, а также на серверах. OMAP4430 — популярный процессор на базе ARM, широко применяемый в смартфонах и планшетных компьютерах.

Скорее всего, вы никогда не слыхали о микроконтроллере ATmega168, но много раз взаимодействовали с ним. Микроконтроллер ATmega168 на базе AVR встречается во многих встроенных системах, от радиочасов до СВЧ-печей. Интерес ко встроенным системам растет, и ATmega168 повсеместно используется благодаря своей исключительно низкой стоимости, широкому выбору программного обеспечения и периферийных устройств, а также изобилию квалифицированных программистов. Безусловно, по количеству установленных экземпляров в мире ATmega168 на порядки опережает процессоры Pentium и Core i3, i5 и i7. ATmega168 устанавливается в одноплатный встроенный компьютер Arduino — популярную систему для энтузиастов, которая была спроектирована в итальянском университете с расчетом на то, чтобы она стоила дешевле обеда в пиццерии.

Многие преподаватели, выстраивающие свои учебные курсы на основе этой книги, просили меня развить тему программирования на языке ассемблера. В шестом издании этот материал был размещен на веб-сайте книги (см. далее), где он легко может дополняться и поддерживаться в актуальном состоянии. В примерах используется ассемблер 8088, который являет собой упрощенную версию невероятно популярного набора команд iA32, используемого процессором Core i7. Можно было выбрать ARM или AVR или другую архитектуру набора команд, о которой мало кто слышал, но у 8088 есть важное преимущество — большинство студентов дома работают на 8088-совместимых системах. Полный набор команд Core i7 слишком сложен для подробного изучения студентами. Набор команд 8088 похож на него, но намного проще.

Кроме того, процессор Core i7, подробно рассмотренный в этом издании книги, способен выполнять программы 8088. Поскольку отлаживать ассемблерный код очень сложно, я разработал несколько инструментальных средств, призван-

ных помочь в процессе изучения языка ассемблера, включая сам ассемблер 8088, а также симулятор и трассер. Эти инструменты работают в средах Windows, UNIX и Linux. Они доступны на веб-сайте книги.

С годами книга увеличилась в объеме (в первом издании было 443 страницы; в этом уже 816). Это неизбежно, поскольку происходит постоянное развитие, и о предмете становится известно все больше и больше. Поэтому если книга используется в целях обучения, нужно иметь в виду, что завершение материала в рамках учебного курса (например, в системе триместров) может оказаться невозможным. Возможный вариант — в качестве минимума изучать первые три главы, часть главы 4 (до раздела 4.4 включительно) и главу 5, а оставшееся время на ваше усмотрение потратить на остаток главы 4, а также отдельные части глав 6, 7 и 8 в зависимости от интересов преподавателя и студентов.

Далее приводится краткая сводка основных изменений по главам (относительно пятого издания). В главе 1 по-прежнему излагается история развития архитектуры компьютеров, в которой объясняется текущее состояние дел и рассматриваются основные вехи пройденного пути. Вероятно, для многих студентов окажется неожиданным тот факт, что самые мощные компьютеры 1960-х годов, стоившие миллионы долларов, по вычислительной мощности не набирают и одного процента от их смартфонов. Обсуждается расширенный спектр существующих компьютеров, включая FPGA, смартфоны, планшеты и игровые приставки. Здесь же представлены архитектуры, выбранные в качестве примера (Core i7, OMAP4430 и ATmega168).

Материал главы 2 был обновлен и переработан. В нее были включены процессоры параллельной обработки данных, включая графические процессоры (GPU). Раздел, посвященный носителям данных, был дополнен описаниями флэш-устройств, которые становятся все более популярными. В раздел ввода-вывода был добавлен новый материал, посвященный современным игровым манипуляторам, включая Wiimote и Kinect, а также сенсорные экраны, используемые в смартфонах и планшетах.

Глава 3 претерпела некоторые изменения. Она, как и прежде, открывается описанием принципа работы транзисторов, чтобы принцип работы современного компьютера был понятен даже студенту, не имеющему представления об аппаратной части. Мы приводим новый материал о программируемых вентильных матрицах (FPGA, Field-Programmable Gate Array) — устройствах, которые снижают стоимость крупномасштабных архитектур уровня логических элементов до такой степени, что последние широко используются в учебных аудиториях. Также здесь представлены высокоуровневые описания трех архитектур, выбранных нами в качестве примера.

Глава 4, в которой изложены основные принципы действия вычислительных систем, осталась после выхода пятого издания почти без изменений. В нее лишь введены три новых раздела, посвященные уровню микроархитектуры процессоров Core 7, OMAP4430 и ATmega168.

Главы 5 и 6 были обновлены для новых архитектур; в частности, появились новые разделы с описаниями наборов команд ARM и AVR. В главе 6 вместо Windows XP материал объясняется на примере Windows 7.

Глава 7, посвященная ассемблеру, осталась практически неизмененной.

Глава 8, наоборот, полностью переписана, и теперь она адекватно отражает развитие параллельных компьютерных архитектур. В нее включена дополнительная информация о многопроцессорной архитектуре Core i7 и подробное описание архитектуры графического процессора общего назначения NVIDIA Fermi. Наконец, разделы, посвященные суперкомпьютерам BlueGene и Red Storm, были приведены в соответствие с последними обновлениями этих гигантских машин.

Глава 9 изменилась. Список литературы был перемещен на веб-сайт, поэтому в главе остались только ссылки, упоминаемые в книге. Многих из них не было в предыдущих изданиях книги. Это вполне естественно; строение компьютера — область компьютерных архитектур динамично развивается.

Приложения А и Б не изменились, да это и понятно — в области двоичных и шестнадцатеричных чисел за последние несколько лет революций не было. Приложение В, посвященное программированию на языке ассемблера, было написано Эвертом Ваттелем (Evert Wattel) из Свободного университета Амстердама. У него огромный опыт обучения работе с инструментарием программирования на ассемблере, и я очень благодарен ему, что он взялся за эту задачу. Материал приложения в целом мало изменился с выхода пятого издания, но программные инструменты теперь размещаются на веб-сайте, а не на прилагаемом компакт-диске.

Помимо инструментов ассемблирования, на веб-сайте есть программа моделирования (графический симулятор), предусмотренная для работы с материалом главы 4. Ее разработчиком стал профессор Ричард Солтер (Richard Salter) из колледжа Оберлин. Ему — отдельное спасибо. Программа поможет студентам лучше понять принципы, представленные в этой главе.

Веб-сайт книги с программами и всем прочим находится по адресу:

http://www.pearsonhighered.com/tanenbaum

После перехода по указанному адресу щелкните на ссылке Companion Website и выберите нужную страницу в появившемся меню. В категорию ресурсов для студентов входят:

- Ассемблер/трассер.
- Графический симулятор.
- Список литературы.

К ресурсам для преподавателей относятся:

- Презентации в формате PowerPoint.
- Решения упражнений, приводимых в конце глав.

Для доступа к преподавательским ресурсам потребуется пароль. Чтобы получить его, преподавателям следует связаться с представителем Pearson Education.

Многие люди читали (отдельные части) рукописи данной книги и высказали ценные замечания и предложения или оказали другую помощь. В частности, мы хотели бы поблагодарить Анну Остин (Anna Austin), Марка Остина (Mark Austin), Ливио Бертакко (Livio Bertacco), Валерию Бертакко (Valeria Bertacco), Дебаприя Чаттерджи (Debapriya Chatterjee), Джейсона Клемонса (Jason Clemons), Эндрю ДеОрио (AndrewDeOrio), Джозефа Грейтхауза (Joseph Greathouse) и Андреа Пеллегрини (Andrea Pellegrini).

Следующие люди прочитали рукопись и предложили свои изменения: Джейсон Д. Бакос (Jason D. Bakos) — университет Южной Каролины, Боб Браун (Bob Brown) — Южный политехнический государственный университет, Эндрю Чен (Andrew Chen) — Государственный университет Миннесоты (Мурхед), Дж. Арчер Харрис (J. Archer Harris) — университет Джеймса Мэдисона, Сьюзан Круке (Susan Krucke) — университет Джеймса Мэдисона, А. Явуз Орук (A. Yavuz Oruc) — университет штата Мэриленд, Фрэнсис Марш (Frances Marsh) — муниципальный колледж Джеймстауна и Крис Шиндлер (Kris Schindler) — университет в Баффало. Спасибо вам всем.

У нас также были помощники, которые участвовали в создании новых упражнений. Это Байрон А. Джефф (Byron A. Jeff) — университет Клейтона, Лора У. Макфолл (Laura W. McFall) — университет ДеПола, Таги М. Мостафави (Taghi M. Mostafavi) — университет Северной Каролины в Шарлотте и Джеймс Нистром (James Nystrom) — университет Ферриса. Им мы также очень благодарны за помощь.

Наш редактор Трэйси Джонсон (Tracy Johnson) была исключительно полезна во многих отношениях, а также проявила большое терпение. Мы высоко ценим работу Кэрол Снайдер (Carole Snyder) по координации работ участников проекта. Боб Энглхарт (Bob Englehardt) прекрасно справился с организацией производственного процесса.

Я (AST) хотел бы поблагодарить Сюзанну за ее любовь и недюжинное терпение. Это уже 21-я книга, и мы до сих пор вместе! Спасибо Барбаре и Марвину за то, что они такие замечательные дети — теперь они знают, чем профессора зарабатывают себе на жизнь. Арон принадлежит уже к следующему поколению: эти дети начинают работать на компьютере еще до того, как идут в детский сад. Натану для этого еще маловат, но после того как он научится ходить, следующим на очереди будет iPad.

Наконец, я (TA) хочу поблагодарить свою тещу Роберту, которая помогла мне с пользой провести время в работе над книгой. Ее столовая в Бассано Дель Граппа (Италия) предоставила столько укрытия, одиночества и вина, сколько было необходимо для решения этой важной задачи.

Эндрю Таненбаум

Тодд Остин

От издателя перевода

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

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Глава 1
Введение

Цифровой компьютер — это машина, которая может решать задачи, исполняя данные ей команды. Последовательность команд, описывающих решение определенной задачи, называется **программой**. Электронные схемы каждого компьютера могут распознавать и исполнять ограниченный набор простых команд. Все программы перед исполнением должны быть превращены в последовательность таких команд, которые обычно не сложнее, чем, например:

- сложить два числа;
- проверить, не является ли число нулем;
- скопировать блок данных из одной части памяти компьютера в другую.

Эти примитивные команды в совокупности составляют язык, на котором люди могут общаться с компьютером. Такой язык называется **машинным**. Разработчик при создании нового компьютера должен решить, какие команды следует включить в машинный язык этого компьютера. Это зависит от назначения компьютера и от задач, которые он должен решать. Обычно стараются сделать машинные команды как можно проще, чтобы избежать сложностей при разработке компьютера и снизить затраты на необходимую электронику. Большинство машинных языков крайне примитивны, из-за чего писать на них и трудно, и утомительно.

Это простое наблюдение с течением времени привело к построению ряда уровней абстракций, каждая из которых надстраивается над абстракцией более низкого уровня. Именно таким образом можно преодолеть сложности и сделать процесс проектирования систематичным и организованным. Мы называем этот подход **многоуровневой компьютерной организацией**. В следующем разделе мы поясним, что этот термин значит. Затем мы расскажем об истории развития проблемы и текущем положении дел, а также рассмотрим некоторые важные примеры.

Многоуровневая компьютерная организация

Как мы уже сказали, существует огромная разница между тем, что удобно людям, и тем, что могут компьютеры. Люди хотят сделать *X*, но компьютеры могут сделать только *Y*. Из-за этого возникает проблема. Цель данной книги — объяснить, как решить эту проблему.

Языки, уровни и виртуальные машины

Вышеупомянутую проблему можно решить двумя способами. Оба способа подразумевают разработку новых команд, более удобных для человека, чем встроенные машинные команды. Эти новые команды в совокупности формируют язык,

который мы будем называть Я1. Встроенные машинные команды тоже формируют язык, и мы будем называть его Я0. Компьютер может исполнять только программы, написанные на его машинном языке Я0. Два способа решения проблемы отличаются тем, каким образом компьютер будет исполнять программы, написанные на языке Я1 — ведь, в конечном итоге, компьютеру доступен только машинный язык Я0.

Первый способ исполнения программы, написанной на языке Я1, подразумевает замену каждой команды эквивалентным набором команд на языке Я0. В этом случае компьютер исполняет новую программу, написанную на языке Я0, вместо старой программы, написанной на Я1. Эта технология называется **трансляцией**.

Второй способ заключается в создании на языке Я0 программы, получающей в качестве входных данных программы, написанные на языке Я1. При этом каждая команда языка Я1 обрабатывается поочередно, после чего сразу исполняется эквивалентный ей набор команд языка Я0. Эта технология не требует составления новой программы на Я0. Она называется **интерпретацией**, а программа, которая осуществляет интерпретацию, называется **интерпретатором**.

Между трансляцией и интерпретацией много общего. В обоих случаях компьютер в конечном итоге исполняет набор команд на языке Я0, эквивалентных командам Я1. Отличие лишь в том, что при трансляции вся программа Я1 переделывается в программу Я0, программа Я1 отбрасывается, а новая программа на Я0 загружается в память компьютера и затем исполняется. Во время выполнения сгенерированная программа на Я0 управляет работой компьютера.

При интерпретации каждая команда программы на Я1 перекодируется в Я0 и сразу же исполняется. Транслированная программа при этом не создается. Работой компьютера управляет интерпретатор, для которого программа на Я1 есть не что иное, как «сырые» входные данные. Оба подхода широко используются как вместе, так и по отдельности.

Впрочем, чем мыслить категориями трансляции и интерпретации, гораздо проще представить себе существование гипотетического компьютера или **виртуальной машины**, для которой машинным языком является язык Я1. Назовем такую виртуальную машину М1, а виртуальную машину для работы с языком Я0 — М0. Если бы такую машину М1 можно было бы сконструировать без больших денежных затрат, язык Я0, да и машина, которая исполняет программы на языке Я0, были бы не нужны. Можно было бы просто писать программы на языке Я1, а компьютер сразу бы их исполнял. Даже с учетом того, что создать виртуальную машину, возможно, не удастся (из-за чрезмерной дороговизны или трудностей разработки), люди вполне могут писать ориентированные на нее программы. Эти программы будут транслироваться или интерпретироваться программой, написанной на языке Я0, а сама она могла бы исполняться существующим компьютером. Другими словами, можно писать программы для виртуальных машин так, как будто эти машины реально существуют.

Трансляция и интерпретация целесообразны лишь в том случае, если языки Я0 и Я1 не слишком отличаются друг от друга. Часто это значит, что язык Я1 хотя и лучше, чем Я0, но все же далек от идеала. Возможно, это несколько обескураживает в свете первоначальной цели создания языка Я1 — освободить про-

граммиста от бремени написания программ на языке, понятным компьютеру, но малоприспособленном для человека. Однако ситуация не так безнадежна.

Очевидное решение проблемы — создание еще одного набора команд, которые в большей степени ориентированы на человека и в меньшей степени на компьютер, чем Я1. Этот третий набор команд также формирует язык, который мы будем называть Я2, а соответствующую виртуальную машину — М2. Человек может писать программы на языке Я2, как будто виртуальная машина для работы с машинным языком Я2 действительно существует. Такие программы могут либо транслироваться на язык Я1, либо исполняться интерпретатором, написанным на языке Я1.

Изобретение целого ряда языков, каждый из которых более удобен для человека, чем предыдущий, может продолжаться до тех пор, пока мы не дойдем до подходящего нам языка. Каждый такой язык использует своего предшественника как основу, поэтому мы можем рассматривать компьютер в виде ряда **уровней**, изображенных на рис. 1.1. Язык, находящийся в самом низу иерархической структуры — самый примитивный, а тот, что расположен на ее вершине — самый сложный.

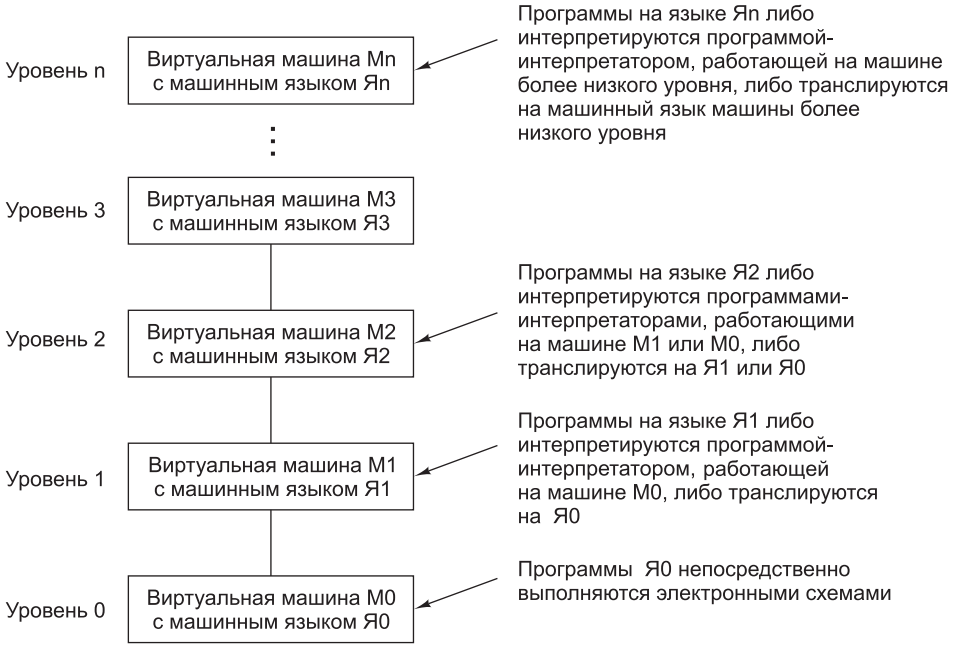


Рис. 1.1. Многоуровневая машина

Между языком и виртуальной машиной существует важная зависимость. Каждая машина поддерживает какой-то определенный машинный язык, состоящий из всех команд, которые эта машина может исполнять. В сущности, машина определяет язык. Аналогичным образом язык определяет машину, а именно машину, которая может исполнять все программы, написанные на этом языке. Конечно, машину, определяемую тем или иным языком, очень сложно и дорого конструировать из электронных схем, однако представить себе такую машину

мы можем. Компьютер, у которого машинным языком был бы C или C++, оказался бы слишком сложным, но в принципе его можно разработать, учитывая высокий уровень современных технологий. Однако существуют веские причины не создавать такой компьютер — такое решение неэффективно по сравнению с другими. Действительно, технология должна быть не только осуществимой, но и рациональной.

Компьютер с *n* уровнями можно рассматривать как *n* разных виртуальных машин, у каждой из которых есть свой машинный язык. Термины «уровень» и «виртуальная машина» мы будем использовать как синонимы. Пожалуйста, учтите, что термин «виртуальная машина», как и многие термины в компьютерной области, также имеет и другие значения; одно из них будет рассмотрено нами позднее в книге. Только программы, написанные на Я0, могут исполняться компьютером без трансляции или интерпретации. Программы, написанные на Я1, Я2, ..., Я*n*, должны проходить через интерпретатор более низкого уровня или транслироваться на язык, соответствующий более низкому уровню.

Человеку, который пишет программы для виртуальной машины уровня *n*, не обязательно знать о трансляторах и интерпретаторах более низких уровней. Машина исполнит эти программы, и не важно, будут они поэтапно исполняться интерпретатором или же их обработает сама машина. В обоих случаях результат один и тот же — это исполнение программы.

Большинству программистов, использующих машину уровня *n*, интересен только самый верхний уровень, который меньше всего сходен с машинным языком. Однако те, кто хочет понять, как в действительности работает компьютер, должны изучить все уровни. Также должны быть знакомы со всеми уровнями разработчики новых компьютеров или новых уровней (то есть новых виртуальных машин). Понятия и технические приемы разработки машин как системы уровней, а также подробное описание этих самых уровней, составляют главный предмет этой книги.

Современные многоуровневые машины

Большинство современных компьютеров состоит из двух и более уровней. Существуют машины даже с шестью уровнями (рис. 1.2). Уровень 0 — это аппаратное обеспечение машины. Его электронные схемы исполняют машинно-зависимые программы уровня 1. Ради полноты нужно упомянуть о существовании еще одного уровня, который расположен ниже нулевого. Этот уровень не показан на рис. 1.2, так как он попадает в сферу электронной техники и, следовательно, не рассматривается в этой книге. Он называется **уровнем физических устройств**. На этом уровне находятся транзисторы, которые для разработчиков компьютеров являются примитивами. Если кого-то заинтересует, как работают транзисторы, ему придется обратиться к области физики твердого тела.

На самом нижнем уровне из тех, что мы будем изучать, а именно на **цифровом логическом уровне**, объекты называются **вентилями**. Хотя вентили строятся из аналоговых компонентов (таких как транзисторы), они могут быть точно смоделированы как цифровые устройства. У каждого вентиля есть одно или несколько цифровых входных данных (сигналов, представляющих 0 или 1). Вентиль вычисляет простые функции этих сигналов, такие как И или ИЛИ.

Рис. 1.2. Шестиуровневый компьютер. Способ поддержки каждого уровня указан под ним, в скобках дано название соответствующего программного обеспечения

Каждый вентиль формируется из нескольких транзисторов. Несколько вентилей формируют 1 бит памяти, который может содержать 0 или 1. Биты памяти, объединенные в группы, например, по 16, 32 или 64, формируют **регистры**. Каждый регистр может содержать одно двоичное число в определенном диапазоне. Из вентилей также может строиться само ядро вычислительной системы. Вентили и цифровой логический уровень подробно рассматриваются в главе 3.

Следующий уровень называется уровнем **микроархитектуры**. На этом уровне находятся наборы из (обычно) 8 или 32 регистров, которые формируют локальную память и схему, называемую **АЛУ (арифметико-логическое устройство)**. АЛУ исполняет простые арифметические операции. Регистры вместе с АЛУ формируют **тракт данных**, по которому поступают данные. Базовая операция тракта данных выполняется следующим образом: выбирается один или два регистра, АЛУ производит над ними какую-либо операцию (например сложение), после чего результат вновь помещается в какой-либо регистр.

На некоторых машинах работа тракта данных контролируется особой программой, которая называется **микропрограммой**. На других машинах тракт данных управляется напрямую аппаратными средствами. В ранних изданиях книги мы назвали этот уровень «уровнем микропрограммирования», потому что раньше на нем почти всегда находился программный интерпретатор. Поскольку сейчас тракт данных обычно контролируется аппаратным обеспечением (по крайней мере частично), мы изменили название, чтобы точнее отразить смысл.

На машинах, где тракт данных контролируется программным обеспечением, микропрограмма — это интерпретатор для команд на уровне 2. Микропрограмма

читает команды из памяти и исполняет их одну за другой, используя при этом тракт данных. Например, при исполнении команды `ADD` она вызывается из памяти, ее операнды помещаются в регистры, АЛУ вычисляет сумму, а затем результат направляется туда, где он должен находиться. На компьютере с аппаратным управлением тракта данных происходит такая же процедура, но при этом нет программы, интерпретирующей команды уровня 2.

Уровень 2 мы будем называть **уровнем архитектуры набора команд**. Каждый производитель публикует руководство для компьютеров, которые он продает, под названием «Руководство по машинному языку *X*», «Принципы работы компьютера *Y*» и т. п. Подобное руководство содержит информацию именно об этом уровне, а не о более низких уровнях. Описываемый в нем набор машинных команд в действительности исполняется микропрограммой-интерпретатором или аппаратным обеспечением. Если производитель поставляет два интерпретатора для одной машины, он должен издать два руководства по машинному языку, отдельно для каждого интерпретатора.

Следующий уровень обычно является гибридным. Большинство команд в его языке есть также и на уровне архитектуры набора команд (команды, имеющиеся на одном из уровней, вполне могут быть представлены и на других уровнях). У этого уровня есть некоторые дополнительные особенности: новый набор команд, другая организация памяти, способность исполнять две и более программ одновременно и некоторые другие. При построении уровня 3 возможно большее разнообразие, чем при построении уровней 1 и 2.

Новые средства, появившиеся на уровне 3, исполняются интерпретатором, который работает на втором уровне. Этот интерпретатор был когда-то назван операционной системой. Команды уровня 3, идентичные командам уровня 2, исполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Другими словами, одна часть команд уровня 3 интерпретируется операционной системой, а другая часть — микропрограммой. Вот почему этот уровень считается гибридным. Мы будем называть этот уровень **уровнем операционной системы**.

Между уровнями 3 и 4 есть принципиальная разница. Нижние три уровня не предназначены для использования рядовыми программистами. Они изначально ориентированы на интерпретаторы и трансляторы, обеспечивающие работу на более высоких уровнях. Эти трансляторы и интерпретаторы создаются **системными программистами**, которые специализируются на разработке новых виртуальных машин. Уровни с четвертого и выше предназначены для прикладных программистов, решающих конкретные задачи.

Еще одно изменение, появившееся на уровне 4, — механизм поддержки более высоких уровней. Уровни 2 и 3 всегда интерпретируются, а уровни 4, 5 и выше обычно (хотя и не всегда) транслируются.

Другое отличие между уровнями 1, 2, 3 и уровнями 4, 5 и выше — специфика языка. Машинные языки уровней 1, 2 и 3 — цифровые. Программы, написанные на этих языках, состоят из длинных рядов цифр, которые воспринимаются компьютерами, но малопонятны для людей. Начиная с уровня 4, языки содержат слова и сокращения, понятные человеку.

Уровень 4 представляет собой символическую форму одного из языков более низкого уровня. На этом уровне человек может писать программы для уров-

ней 1, 2 и 3 в форме не настолько неприятной, как язык виртуальных машин. Эти программы сначала транслируются на язык уровня 1, 2 или 3, а затем интерпретируются соответствующей виртуальной или реально существующей машиной. Программа, которая исполняет трансляцию, называется **ассемблером**.

Уровень 5 обычно состоит из языков, разработанных для прикладных программистов. Такие языки называются **языками высокого уровня**. Существуют сотни языков высокого уровня. Наиболее известные среди них — C, C++, Java, Perl, Python и PHP. Программы, написанные на этих языках, обычно транслируются на уровень 3 или 4. Трансляторы, которые обрабатывают эти программы, называются **компиляторами**, хотя в некоторых случаях имеет место интерпретация. Например, программы на языке Java сначала транслируются на язык, напоминающий машинные команды и называемый байт-кодом Java, который затем интерпретируется.

В некоторых случаях уровень 5 состоит из интерпретатора для конкретной прикладной области, например символической логики. Он предусматривает данные и операции для решения задач в этой области в контексте, хорошо понятном специалисту в этой предметной области.

Итак, из этого описания важно запомнить, что компьютер проектируется как иерархическая структура уровней, которые надстраиваются друг над другом. Каждый уровень представляет собой абстракцию некоторых объектов и операций. Рассматривая и анализируя строение компьютера подобным образом, мы можем не принимать во внимание лишние подробности и, таким образом, сделать сложный предмет более простым для понимания.

Набор типов данных, операций и характеристик каждого отдельно взятого уровня называется **архитектурой**. Архитектура связана с аспектами, видимыми пользователю этого уровня. Например, сведения о том, сколько памяти можно использовать при написании программы, — часть архитектуры. Аспекты реализации (например, технология, применяемая при реализации памяти) не являются частью архитектуры. Изучая методы проектирования программных элементов компьютерной системы, мы изучаем **компьютерную архитектуру**. На практике термины «компьютерная архитектура» и «компьютерная организация» употребляются как синонимы.

Развитие многоуровневых машин

В этом разделе мы кратко расскажем об истории развития многоуровневых машин, покажем, как число и природа уровней менялись с годами. Программы, написанные на машинном языке (уровень 1), могут сразу без применения интерпретаторов и трансляторов исполняться электронными схемами компьютера (уровень 0). Эти электронные схемы вместе с памятью и средствами ввода-вывода формируют **аппаратное обеспечение** компьютера. Аппаратное обеспечение состоит из материальных объектов — интегральных схем, печатных плат, кабелей, источников электропитания, модулей памяти и принтеров. Абстрактные понятия, алгоритмы и команды к аппаратному обеспечению не относятся.

Программное обеспечение, напротив, состоит из **алгоритмов** (подробных последовательностей команд, которые описывают решение некоторой задачи) и их компьютерных представлений, то есть программ. Программы могут храниться на

жестком диске, гибком диске, компакт-диске или других носителях, но это не так уж важно; в сущности, программное обеспечение — это набор команд, составляющих программы, а не физические носители, на которых эти программы записаны.

В самых первых компьютерах граница между аппаратным и программным обеспечением была очевидна. Однако со временем произошло значительное размывание этой границы, в первую очередь благодаря тому, что в процессе развития компьютеров уровни добавлялись, убирались и сливались между собой. В настоящее время очень сложно отделить их друг от друга (Vahid, 2003). Собственно, центральная тема нашей книги может быть сформулирована следующим образом:

Аппаратное и программное обеспечение логически эквивалентно.

Любая операция, исполняемая программным обеспечением, может быть реализована аппаратным обеспечением (желательно после того, как она будет продумана). Как говорила Карен Панетта (Karen Panetta): «Аппаратное обеспечение — это всего лишь окаменевшее программное обеспечение». Конечно, обратное тоже верно: любая команда, исполняемая аппаратным обеспечением, может быть смоделирована программно. Решение о разделении функций аппаратного и программного обеспечения основано на таких факторах, как стоимость, быстродействие, надежность, частота ожидаемых изменений. Незыблемых правил, требующих, чтобы операция *X* была реализована в аппаратном обеспечении, а операция *Y* непременно программировалась, очень мало. Эти решения меняются в зависимости от тенденций экономического и технологического развития.

Изобретение микропрограммирования

У первых цифровых компьютеров 40-х годов было только два уровня: уровень архитектуры набора команд, на котором осуществлялось программирование, и цифровой логический уровень, на котором программы исполнялись. Схемы цифрового логического уровня были ненадежны, сложны для производства и понимания.

В 1951 году Морис Уилкс (Maurice Wilkes), исследователь Кембриджского университета, предложил идею трехуровневого компьютера, призванную радикально упростить аппаратное обеспечение, а следовательно, сократить количество (ненадежных) электронных ламп [Wilkes, 1951]. Эта машина должна была иметь встроенный неизменяемый интерпретатор (микропрограмму), функция которого заключалась в исполнении программ уровня ISA посредством интерпретации. Так как аппаратное обеспечение должно было теперь вместо программ уровня ISA исполнять только микропрограммы с ограниченным набором команд, требовалось меньшее количество электронных схем. Поскольку электронные схемы тогда делались из электронных ламп, данное упрощение призвано было сократить количество ламп и, следовательно, повысить надежность (которая в то время выражалась числом поломок за день).

В 50-е годы было построено несколько трехуровневых машин. В 60-х годах число таких машин значительно увеличилось. К 70-м годам идея о том, что написанная программа сначала должна интерпретироваться микропрограммой, а не исполняться непосредственно электроникой, стала преобладающей. В наши дни она используется всеми современными компьютерами.

Изобретение операционной системы

В те времена, когда компьютеры только появились, принципы работы с ними сильно отличались от современных. Одним компьютером пользовалось большое количество людей. Рядом с машиной лежал листок бумаги, и если программист хотел запустить свою программу, он записывался на какое-то определенное время, скажем, на среду с 3 часов ночи до 5 утра (многие программисты любили работать в тишине). В назначенное время программист направлялся в комнату, где стояла машина, с пачкой перфокарт (которые тогда служили средством ввода) в одной руке и хорошо заточенным карандашом в другой. Каждая перфокарта содержала 80 колонок; на ней в определенных местах пробивались отверстия. Войдя в комнату, программист вежливо просил предыдущего программиста освободить место и приступал к работе.

Если он хотел запустить программу на языке FORTRAN, ему необходимо было пройти следующие этапы:

1. Он подходил к шкафу, где находилась библиотека программ, брал большую зеленую стопку перфокарт с надписью «Компилятор FORTRAN», помещал их в считывающее устройство и нажимал кнопку «Пуск».
2. Он помещал стопку карточек со своей программой, написанной на языке FORTRAN, в считывающее устройство и нажимал кнопку «Продолжить». Программа считывалась.
3. Когда компьютер прекращал работу, программист считывал свою программу во второй раз. Некоторые компиляторы требовали только одного считывания перфокарт, но в большинстве случаев необходимо было производить эту процедуру несколько раз. Каждый раз требовалось загружать большую стопку перфокарт.
4. В конце концов трансляция завершалась. Программист часто начинал нервничать, потому что если компилятор находил ошибку в программе, ему приходилось исправлять ее и начинать процесс ввода программы заново. Если ошибок не было, компилятор выдавал в виде перфокарт программу на машинном языке.
5. Тогда программист помещал эту программу на машинном языке в устройство считывания вместе с пачкой перфокарт из библиотеки подпрограмм и загружал обе эти программы.

Начиналось исполнение программы. В большинстве случаев она не работала, неожиданно останавливаясь в середине. Обычно в этом случае программист начинал дергать переключатели на пульте и смотрел на лампочки. В случае удачи он находил и исправлял ошибку, подходил к шкафу, в котором лежала большая зеленая стопка перфокарт с надписью «Компилятор FORTRAN», и начинал все заново. В случае неудачи он делал распечатку содержания памяти, которая называлась **дампом оперативной памяти**, и брал эту распечатку домой для изучения.

Эта процедура с небольшими вариациями была обычной во многих компьютерных центрах на протяжении многих лет. Программистам приходилось учиться, как работать с машиной и что нужно делать, если она выходила из строя, — а происходило это довольно часто. Машина постоянно простаивала без работы, пока люди носили перфокарты по комнате или ломали головы над тем, почему программа не работает.

В 60-е годы человек попытался ускорить дело, автоматизировав работу оператора. Программа под названием **операционная система** загружалась в компьютер на все время его работы. Программист приносил пачку перфокарт со специализированной программой, которая исполнялась операционной системой. На рис. 1.3 показан пример задания для первой широко распространенной операционной системы FMS (FORTRAN Monitor System) к компьютеру 709 компании IBM.

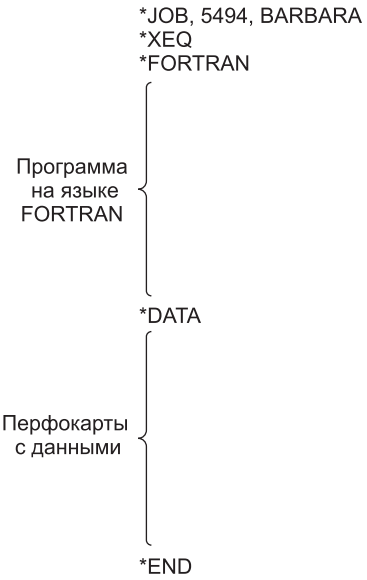


Рис. 1.3. Схема работы с операционной системой FMS

Операционная система считывала перфокарту *JOB и использовала содержащуюся на ней информацию для учета системных ресурсов (звездочка ставилась, чтобы отличать перфокарты с управляющей программой от перфокарт с данными). Затем операционная система считывала перфокарту *FORTRAN — команду для загрузки компилятора FORTRAN с магнитной ленты. После этого компилятор считывал и компилировал программу, написанную на языке FORTRAN. Как только компилятор заканчивал работу, операционная система считывала перфокарту *DATA — команду по исполнению транслированной программы с использованием перфокарт данных.

Хотя операционная система была придумана для того, чтобы автоматизировать работу оператора (отсюда и название), она стала первым шагом в развитии новой виртуальной машины. Перфокарту *FORTRAN можно рассматривать как виртуальную команду «откомпилировать программу», а перфокарту *DATA — как виртуальную команду «выполнить программу». И хотя этот уровень состоял всего из двух команд, он стал первым шагом в развитии виртуальных машин.

В последующие годы операционные системы все больше и больше усложнялись. К уровню архитектуры набора команд добавлялись новые команды, приспособления и функции, из которых в конечном итоге сформировался

новый уровень. Некоторые команды нового уровня были идентичны командам предыдущего, но некоторые (в частности, команды ввода-вывода) полностью отличались. Эти новые команды тогда назывались **макросами операционной системы**, или **вызовами супервизора**. Сейчас обычно используется термин **системный вызов**.

Первые операционные системы считывали пачки перфокарт и распечатывали результат на принтере. Такая организация вычислений называлась **пакетным режимом**. Чтобы получить результат, обычно приходилось ждать несколько часов. При таких условиях было трудно развивать программное обеспечение.

В начале 60-х годов исследователи из Дартмутского колледжа, Массачусетского технологического института (МТИ) разработали операционную систему, которая давала возможность работать с компьютером сразу нескольким программистам. В этой системе к центральному компьютеру через телефонные линии подсоединялись отдаленные терминалы. Таким образом, центральный процессор разделялся между большим количеством пользователей. Программист мог напечатать свою программу и получить результаты почти сразу прямо в офисе, гараже или где бы то ни было еще (там, где находился терминал). Эти системы назывались (и сейчас называются) **системами разделения времени**.

Нас интересуют только те компоненты операционной системы, которые интерпретируют команды уровня 3, отсутствующие на уровне ISA, а не возможности разделения времени. Хотя в дальнейшем не будем особо подчеркивать этот факт, необходимо понимать, что интерпретация возможностей, добавившихся на уровне ISA, — не единственная функция операционных систем.

Смещение функциональности на уровень микрокода

С 1970 года, когда получило развитие микропрограммирование, производители осознали, что теперь новые машинные команды можно добавлять простым расширением микропрограммы. Иначе говоря, они могли добавлять «аппаратное обеспечение» (новые команды) путем программирования. Это открытие буквально привело к взрыву в производстве наборов машинных команд, поскольку производители начали конкурировать друг с другом — каждый старался, чтобы его набор команд был больше и лучше, чем у других. Многие команды не представляли особой ценности, поскольку те же задачи можно было легко решить, используя уже существующие команды, но обычно они выполнялись немного быстрее. Например, во многих компьютерах использовалась команда `INC` (INCrement), которая прибавляла к числу единицу. Тогда уже существовала общая команда сложения `ADD`, и не было необходимости вводить новую команду, прибавляющую к числу единицу. Тем не менее команда `INC` работала немного быстрее, чем `ADD`, поэтому ее также включили в набор команд.

Многие команды добавлялись в микропрограмму по той же причине. Среди них можно назвать команды для:

- умножения и деления целых чисел;
- арифметических действий над числами с плавающей точкой;
- вызова и прекращения действия процедур;
- ускорения циклов;
- работы с символьными строками.

Как только производители поняли, что добавлять новые команды очень легко, они начали думать, какими дополнительными техническими возможностями можно наделить микропрограмму. Приведем несколько примеров:

- ускорение работы с массивами (индексная и косвенная адресация);
- перемещение программы из одного раздела памяти в другой после запуска программы (переадресация);
- системы прерывания, которые дают сигнал процессору, как только закончена операция ввода или вывода;
- способность приостановить одну программу и начать другую, используя небольшое число команд (переключение процесса);
- специальные команды для обработки изображений, звуковых и мультимедийных данных.

За последующие годы добавилось много других команд и технических средств, обычно ускорявших выполнение некоторой конкретной операции.

Конец микропрограммирования

В 60-х–70-х годах количество микропрограмм значительно увеличилось. Однако они работали все медленнее и медленнее, поскольку занимали все больше места. В конце концов исследователи осознали, что отказ от микропрограмм резко сократит количество команд, и компьютеры станут работать быстрее. Таким образом, компьютеры вернулись к тому состоянию, в котором они находились до изобретения микропрограммирования.

Впрочем, нельзя сказать, что эта ветвь привела в тупик. Современные процессоры продолжают использовать микропрограммы для преобразования сложных команд во внутренний микрокод, который может напрямую выполняться на оптимизированных аппаратных компонентах.

Мы рассмотрели развитие компьютеров, чтобы показать, что граница между аппаратным и программным обеспечением постоянно смещается. Сегодняшнее программное обеспечение может быть завтрашним аппаратным обеспечением, и наоборот. Более того, также обстоит дело и с уровнями — между ними нет четких границ. Для программиста не важно, как на самом деле исполняется команда (за исключением, может быть, скорости исполнения). Программист, работающий на уровне архитектуры набора команд, может использовать команду умножения, как будто это аппаратная команда, и даже не задумываться об этом. То, что для одного человека — программное обеспечение, для другого — аппаратное. Позже мы еще вернемся к этим вопросам.

Развитие компьютерной архитектуры

В ходе эволюции компьютерных технологий были разработаны сотни разных компьютеров. Многие из них давно забыты, в то время как влияние других на современные идеи оказалось весьма значительным. В этом разделе мы дадим краткий обзор некоторых ключевых исторических моментов, чтобы лучше понять, каким образом разработчики дошли до концепции современных компьютеров. Разумеется, мы рассмотрим только основные моменты развития,

оставив многие подробности за скобками. Некоторые «исторические вехи» этого пути — компьютеры, которые мы будем рассматривать — представлены в табл. 1.1. Хорошую подборку дополнительного исторического материала по «отцам-основателям» компьютерной эры можно найти у Слейтера (1987). Краткие биографии с красивыми цветными фотографиями ключевых фигур, выполненными Луисом-Фабианом Бакрахом, представлены в подарочном альбоме Моргана (1997).

Таблица 1.1. Основные этапы развития компьютеров

Год выпуска	Название компьютера	Создатель	Примечания
1834	Аналитическая машина	Бэббидж	Первая попытка построить цифровой компьютер
1936	Z1	Зус	Первая релейная вычислительная машина
1943	COLOSSUS	Британское правительство	Первый электронный компьютер
1944	Mark I	Айкен	Первый американский компьютер общего назначения
1946	ENIAC I	Экерт/ Моушли	С этой машины начинается история современных компьютеров
1949	EDSAC	Уилкс	Первый компьютер с программами, хранящимися в памяти
1951	Whirlwind I	МТИ	Первый компьютер реального времени
1952	IAS	Фон Нейман	Эта архитектура используется в большинстве современных компьютеров
1960	PDP-1	DEC	Первый мини-компьютер (продано 50 экземпляров)
1961	1401	IBM	Очень популярный компьютер для малого бизнеса
1962	7094	IBM	Лидер в области научных расчетов начала 1960-х годов
1963	B5000	Burroughs	Первая машина, разработанная для языка высокого уровня
1964	360	IBM	Первое семейство компьютеров
1964	6600	CDC	Первый суперкомпьютер для научных расчетов
1965	PDP-8	DEC	Первый мини-компьютер массового потребления (продано 50 000 экземпляров)
1970	PDP-11	DEC	Эти мини-компьютеры доминировали на компьютерном рынке в 70-е годы
1974	8080	Intel	Первый универсальный 8-разрядный компьютер на микросхеме

Год выпуска	Название компьютера	Создатель	Примечания
1974	CRAY-1	Cray	Первый векторный суперкомпьютер
1978	VAX	DEC	Первый 32-разрядный супермини-компьютер
1981	IBM PC	IBM	Началась эра современных персональных компьютеров
1981	Osborne-1	Osborne	Первый портативный компьютер
1983	Lisa	Apple	Первый ПК с графическим пользовательским интерфейсом
1985	386	Intel	Первый 32-разрядный предшественник линейки Pentium
1985	MIPS	MIPS	Первый компьютер RISC
1985	XC2064	Xilinx	Первая программируемая вентильная матрица (FPGA)
1987	SPARC	Sun	Первая рабочая станция RISC на основе процессора SPARC
1989	GridPad	Grid Systems	Первый коммерческий планшетный компьютер
1990	RS6000	IBM	Первый суперскалярный компьютер
1992	Alpha	DEC	Первый 64-разрядный ПК
1992	Simon	IBM	Первый смартфон
1993	Newton	Apple	Первый карманный компьютер
2001	POWER4	IBM	Первая двухъядерная многопроцессорная микросхема

Нулевое поколение — механические компьютеры (1642–1945)

Первым человеком, создавшим счетную машину, был французский ученый Блез Паскаль (1623–1662), в честь которого назван один из языков программирования. Паскаль сконструировал эту машину в 1642 году, когда ему было всего 19 лет, для своего отца, сборщика налогов. Это была механическая конструкция с шестеренками и ручным приводом.

Счетная машина Паскаля могла исполнять только операции сложения и вычитания, но тридцать лет спустя великий немецкий математик барон Готфрид Вильгельм фон Лейбниц (1646–1716) построил другую механическую машину, которая помимо сложения и вычитания могла исполнять операции умножения и деления. В сущности, Лейбниц три века назад создал подобие карманного калькулятора с четырьмя функциями.

Еще через 150 лет профессор математики Кембриджского университета, Чарльз Бэббидж (1792–1871), изобретатель спидометра, разработал и сконструи-

ровал **разностную машину**. Эта механическая машина, которая, как и машина Паскаля, могла только складывать и вычитать, подсчитывала таблицы чисел для морской навигации. В машину был заложен только один алгоритм — метод конечных разностей с использованием полиномов. У этой машины был довольно интересный способ вывода информации: результаты выдавливались стальным штампом на медной дощечке, что предвосхитило более поздние средства ввода-вывода с одноразовой записью — перфокарты и компакт-диски.

Хотя его устройство работало довольно неплохо, Бэббиджу вскоре наскучила машина, исполнявшая только один алгоритм. Он потратил очень много времени, большую часть своего семейного состояния и еще 17 000 фунтов, выделенных правительством, на разработку **аналитической машины**. У аналитической машины было 4 компонента: запоминающее устройство (память), вычислительное устройство, устройство ввода (для считывания перфокарт), устройство вывода (перфоратор и печатающее устройство). Память состояла из 1000 слов по 50 десятичных разрядов; каждое из слов содержало переменные и результаты. Вычислительное устройство принимало операнды из памяти, затем исполняло операции сложения, вычитания, умножения или деления и возвращало полученный результат обратно в память. Как и разностная машина, это устройство было механическим.

Преимущество аналитической машины заключалось в том, что она могла исполнять разные задания. Она считывала команды с перфокарт и исполняла их. Некоторые команды приказывали машине взять два числа из памяти, перенести их в вычислительное устройство, произвести над ними операцию (например, сложить) и отправить результат обратно в запоминающее устройство. Другие команды проверяли число, а иногда совершали операцию перехода в зависимости от того, положительное оно или отрицательное. Если в считывающее устройство вводились перфокарты с другой программой, то машина исполняла другой набор операций. То есть в отличие от разностной аналитическая машина могла выполнять несколько алгоритмов.

Поскольку аналитическая машина программировалась на элементарном ассемблере, ей было необходимо программное обеспечение. Чтобы создать это программное обеспечение, Бэббидж нанял молодую женщину — Аду Августу Лавлейс (Ada Augusta Lovelace), дочь знаменитого британского поэта Байрона. Ада Лавлейс была первым в мире программистом. В ее честь назван современный язык программирования — Ada.

К несчастью, подобно многим современным инженерам, Бэббидж так никогда и не отладил свой компьютер. Ему нужны были тысячи и тысячи шестеренок, сделанных с такой точностью, которая в 19 веке была недоступна. Но идеи Бэббиджа опередили его эпоху, и даже сегодня большинство современных компьютеров по конструкции сходны с аналитической машиной. Поэтому справедливо будет сказать, что Бэббидж был дедушкой современного цифрового компьютера.

В конце 30-х годов немецкий студент Конрад Зус (Konrad Zuse) сконструировал несколько автоматических счетных машин с использованием электромагнитных реле. Ему не удалось получить денежные средства от правительства на свои разработки, потому что началась война. Зус ничего не знал о работе Бэббиджа, его машины были уничтожены во время бомбежки Берлина в 1944 году, поэто-

му его работа никак не повлияла на будущее развитие компьютерной техники. Однако он был одним из пионеров в этой области.

Немного позже в Америке конструированием счетных машин занялись два человека: Джон Атанасов (John Atanasoff) из Колледжа штата Айова и Джордж Стиббиц из Bell Labs. Машина Атанасова была чрезвычайно мощной для того времени. В ней использовалась двоичная арифметика и память на базе конденсаторов, которые периодически обновлялись, чтобы избежать утечки заряда. Современная динамическая память (ОЗУ) работает точно по такому же принципу. К несчастью, эта машина так и не стала действующей. В каком-то смысле Атанасов был похож на Бэббиджа: провидец, мечты которого в конечном итоге разбились о несовершенство технологий своего времени.

Компьютер Джорджа Стиббица (George Stibbitz) действительно работал, хотя и был примитивнее, чем машина Атанасова. Стиббиц продемонстрировал свою машину на конференции в Дартмутском колледже в 1940 году. На этой конференции присутствовал Джон Моушли (John Mauchley), ничем не примечательный на тот момент профессор физики из университета Пенсильвании. Позднее он стал очень известным в области компьютерных разработок.

Пока Зус, Стиббицс и Атанасов разрабатывали автоматические счетные машины, молодой Говард Айкен (Howard Aiken) в Гарварде упорно выполнял нудные ручные вычисления для своей докторской диссертации. После получения степени Айкен осознал важность автоматизации вычислений. Он пошел в библиотеку, прочитал о работе Бэббиджа и решил создать из реле такой же компьютер, который Бэббиджу не удалось создать из зубчатых колес.

Работа над первым компьютером Айкена «Mark I» была закончена в Гарварде в 1944 году. Компьютер имел 72 слова по 23 десятичных разряда каждое, а время выполнения операции составляло 6 секунд. В устройствах ввода-вывода использовалась перфолента. К тому времени, как Айкен закончил работу над компьютером «Mark II», релейные компьютеры уже устарели. Началась эра электроники.

Первое поколение — электронные лампы (1945–1955)

Стимулом к созданию электронного компьютера стала Вторая мировая война. В начале войны германские подводные лодки наносили серьезный ущерб британскому флоту. Германские адмиралы посылали на подводные лодки по радио команды, и хотя англичане могли перехватывать эти команды, проблема была в том, что радиограммы были закодированы с помощью прибора под названием **ENIGMA**, предшественник которого был спроектирован изобретателем-дилетантом и бывшим президентом США Томасом Джефферсоном.

В начале войны англичанам удалось приобрести ENIGMA у поляков, которые, в свою очередь, украли ее у немцев. Однако чтобы расшифровать закодированное послание, требовалось огромное количество вычислений, и их нужно было произвести сразу после перехвата радиограммы. Поэтому британское правительство основало секретную лабораторию для создания электронного компьютера под названием COLOSSUS. В создании этой машины принимал участие знаменитый британский математик Алан Тьюринг. COLOSSUS работал уже в 1943 году, но так как британское правительство полностью контролировало этот проект и рас-

сматривало его как военную тайну на протяжении 30 лет, COLOSSUS не стал базой для дальнейшего развития компьютеров. Мы упомянули о нем только потому, что это был первый в мире электронный цифровой компьютер.

Кроме уничтожения машины Зуса и стимула для создания COLOSSUS, война повлияла и на развитие компьютерной техники в США. Армии нужны были таблицы, которые использовались при наведении тяжелой артиллерии. Сотни женщин нанимались для расчетов на ручных счетных машинах и заполнения полей этих таблиц (считалось, что женщины аккуратнее в расчетах, чем мужчины). Тем не менее этот процесс требовал много времени, и в нем часто случались ошибки.

Джон Моушли, который был знаком с работами Атанасова и Стиббица, понимал, что армия заинтересована в счетных машинах. Он обратился к армии с заявкой на финансирование работ по созданию электронного компьютера. Заявка было удовлетворена в 1943 году, и Моушли со своим студентом Дж. Преспером Экертом (J. Presper Eckert) начали конструировать электронный компьютер, который они назвали **ENIAC** (Electronic Numerical Integrator and Computer — электронный цифровой интегратор и калькулятор). ENIAC состоял из 18 000 электровакуумных ламп и 1500 реле, весил 30 тон и потреблял 140 киловатт электроэнергии. У машины было 20 регистров, каждый из которых мог содержать 10-разрядное десятичное число. (Десятичный регистр — это память очень маленького объема, которая может вмещать число до какого-либо определенного максимального количества разрядов, что-то вроде одометра, запоминающего километраж пройденного автомобилем пути.) Программирование ENIAC осуществлялось при помощи 6000 многоканальных переключателей и многочисленных кабелей, подключавшихся к разъемам.

Работа над машиной была закончена в 1946 году, когда она уже была не нужной — по крайней мере, для достижения первоначально поставленных целей. Поскольку война закончилась, Моушли и Экерту позволили организовать школу, где они рассказывали о своей работе коллегам-ученым. В этой школе и зародился интерес к созданию больших цифровых компьютеров.

После появления школы за конструирование электронных вычислительных машин взялись другие исследователи. Первым рабочим компьютером был EDSAC (1949 год). Эту машину сконструировал Морис Уилкс в Кембриджском университете. Далее — JOHNNIAC в корпорации Rand, ILLIAC в университете Иллинойса, MANIAC в лаборатории Лос-Аламоса и WEIZAC в институте Вайцмана в Израиле.

Экерт и Моушли вскоре начали работу над машиной **EDVAC** (Electronic Discrete Variable Computer — электронная дискретная параметрическая машина). К несчастью, этот проект закрылся, когда они ушли из университета, чтобы основать компьютерную корпорацию в Филадельфии (Силиконовой долины тогда еще не было). После ряда слияний эта компания превратилась в Unisys Corporation.

Экерт и Моушли хотели получить патент на изобретение цифровой вычислительной машины. Оглядываясь назад, можно сказать, что иметь такой патент было бы неплохо. После нескольких лет судебной тяжбы было вынесено решение, что патент недействителен, так как цифровую вычислительную машину

изобрел Атанасов, а поскольку он не оформил патент, изобретение фактически стало общественным достоянием.

В то время как Экерт и Моушли работали над машиной EDVAC, один из участников проекта ENIAC, Джон фон Нейман, поехал в Институт специальных исследований в Принстоне, чтобы сконструировать собственную версию EDVAC под названием **IAS** (Immediate Address Storage — память с прямой адресацией). Фон Нейман был гением того же уровня, что и Леонардо да Винчи. Он знал много языков, был специалистом в физике и математике, обладал феноменальной памятью: он помнил все, что когда-либо слышал, видел или читал. Он мог дословно процитировать по памяти текст книг, которые читал несколько лет назад. Когда фон Нейман стал интересоваться вычислительными машинами, он уже был самым знаменитым математиком в мире.

Фон Нейман вскоре осознал, что программирование компьютеров с большим количеством переключателей и кабелей — занятие медленное, утомительное и неудобное. Он пришел к мысли, что программа должна быть представлена в памяти компьютера в цифровой форме, вместе с данными. Он также отметил, что десятичная арифметика, используемая в машине ENIAC, где каждый разряд представлялся десятью электронными лампами (1 включена и 9 выключены), может быть заменена параллельной двоичной арифметикой. Атанасов пришел к аналогичному выводу на несколько лет раньше.

Основной проект, который фон Нейман описал вначале, известен сейчас как **фон-неймановская вычислительная машина**. Он был использован в EDSAC, первой машине с программой в памяти, и даже сейчас, более чем полвека спустя, является основой большинства современных цифровых компьютеров. Сам замысел и машина IAS, построенная при участии Германа Голдстайна (Herman Goldstine), оказали очень большое влияние на дальнейшее развитие компьютерной техники, поэтому стоит кратко описать проект фон Неймана. Стоит иметь в виду, что хоть проект всегда ассоциируется с именем фон Неймана, в его разработке приняли деятельное участие другие ученые — в частности, Голдстайн. Архитектура этой машины представлена на рис. 1.4.

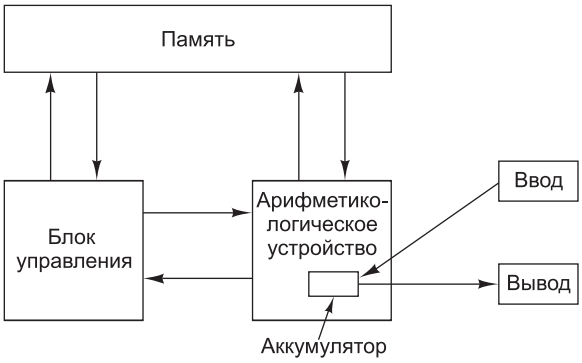


Рис. 1.4. Схема фон-неймановской вычислительной машины

Машина фон Неймана состояла из пяти основных частей: памяти, арифметико-логического устройства, устройства управления, а также устройств ввода-вывода. Память состояла из 4096 слов, каждое слово содержало 40 бит (0 или 1). Каждое

слово содержало или 2 команды по 20 бит, или целое 40-разрядное число со знаком на 40 бит. 8 бит определяли тип команды, а остальные 12 бит определяли одно из 4096 слов памяти. Арифметический блок и блок управления составляли «мозговой центр» компьютера. В современных машинах эти блоки сочетаются в одной микросхеме, называемой **центральным процессором (ЦП)**.

Внутри арифметико-логического устройства находился особый внутренний регистр на 40 бит, так называемый **аккумулятор**. Типичная команда прибавляла слово из памяти к аккумулятору или сохраняла содержимое аккумулятора в памяти. Эта машина не исполняла арифметические операции с плавающей точкой, поскольку фон Нейман считал, что любой сведущий математик способен держать дробную часть в уме.

Примерно в то же время, когда фон Нейман работал над машиной IAS, исследователи МТИ разрабатывали свой компьютер Whirlwind I. В отличие от IAS, ENIAC и других машин того же типа со словами большой длины, предназначенными для серьезных вычислений, машина Whirlwind I имела слова по 16 бит и предназначалась для работы в реальном времени. Этот проект привел к изобретению Джеем Форрестером (Jay Forrester) памяти на магнитном сердечнике, а затем и первого серийного мини-компьютера.

В то время IBM была маленькой компанией, производившей перфокарты и механические машины для сортировки перфокарт. Хотя фирма IBM частично финансировала проект Айкена, она не интересовалась компьютерами и только в 1953 году построила компьютер 701, через много лет после того, как компания Экерта и Моушли со своим компьютером UNIVAC стала номером один на компьютерном рынке. В 701 было 2048 слов по 36 бит, каждое слово содержало две команды. Это была первая машина из серии, которая заняла лидирующее положение на рынке в течение ближайших десяти лет. Через три года появился компьютер 704, у которого было 4096 слов памяти, команды по 36 бит и процессор для вычислений с плавающей точкой. В 1958 году компания IBM начала работу над последним компьютером на электронных лампах, 709, который по сути представлял собой усовершенствованную версию 704.

Второе поколение — транзисторы (1955–1965)

Транзистор был изобретен сотрудниками лаборатории Bell Laboratories Джоном Бардином (John Bardeen), Уолтером Браттейном (Walter Brattain) и Уильямом Шокли (William Shockley), за что в 1956 году они получили Нобелевскую премию в области физики. В течение десяти лет транзисторы произвели революцию в производстве компьютеров, и к концу 50-х годов компьютеры на вакуумных лампах стали пережитком прошлого. Первый компьютер на транзисторах был построен в лаборатории МТИ. Он содержал слова из 16 бит, как и Whirlwind I. Компьютер назывался **TX-0** (Transistorized eXperimental computer 0 — экспериментальная транзисторная вычислительная машина 0) и предназначался только для тестирования будущей машины TX-2.

Машина TX-2 не имела большого значения, но один из инженеров из этой лаборатории, Кеннет Ольсен (Kenneth Olsen), в 1957 году основал компанию DEC (Digital Equipment Corporation — корпорация по производству цифровой аппаратуры) для производства серийной машины, сходной с TX-0. Эта машина,

PDP-1, появилась только через четыре года главным образом потому, что те, кто финансировал DEC, считали, что у производства компьютеров нет будущего. В конце концов, Т. Дж. Уотсон, бывший президент IBM, однажды сказал, что мировой рынок компьютеров составляет четыре или пять единиц. Поэтому компания DEC продавала в основном небольшие электронные платы.

Компьютер PDP-1 появился только в 1961 году. Он имел 4096 слов по 18 бит и быстродействие 200 000 команд в секунду. Этот параметр был в два раза меньше, чем у 7090, транзисторного аналога 709 и самого быстрого компьютера в мире на то время. Но PDP-1 стоил 120 000 долларов, в то время как 7090 стоил миллионы. Компания DEC продала десятки компьютеров PDP-1, и так появилась компьютерная промышленность.

Одну из первых машин модели PDP-1 отдали в МТИ, где она сразу привлекла внимание некоторых молодых исследователей, подающих большие надежды. Одним из нововведений PDP-1 был дисплей с размером 512 × 512 пикселов, на котором можно было рисовать точки. Вскоре студенты МТИ составили специальную программу для PDP-1, чтобы играть в «Космическую войну» — первую в мире компьютерную игру.

Через несколько лет компания DEC разработала модель PDP-8, 12-разрядный компьютер. PDP-8 стоил гораздо дешевле, чем PDP-1 (16 000 долларов). Главное нововведение — единственная шина (omnibus), показанная на рис. 1.5. **Шина** — это набор параллельно соединенных проводов, связывающих компоненты компьютера. Это нововведение радикально отличало PDP-8 от IAS. Такая архитектура с тех пор стала использоваться во всех малых компьютерах. Компания DEC продала 50 000 компьютеров модели PDP-8 и стала лидером на рынке мини-компьютеров.

Рис. 1.5. Шина компьютера PDP-8

Как уже отмечалось, с изобретением транзисторов компания IBM построила транзисторную версию 709 — 7090, а позднее — 7094. У этой версии время цикла составляло 2 микросекунды, а память состояла из 32 536 слов по 36 бит. 7090 и 7094 были последними компьютерами типа ENIAC, но они занимали ведущее положение в области научных расчетов в 60-х годах прошлого века.

В то же время компания IBM зарабатывала большие деньги на продаже небольших компьютеров 1401 для коммерческих расчетов. Эта машина могла считывать и записывать магнитные ленты и перфокарты и распечатывать результат так же быстро, как и 7094, но при этом стоила дешевле. Для научных вычислений она не подходила, но зато была очень удобна для коммерческого учета.

Архитектура 1401 была необычной тем, что в ней не было регистров и даже фиксированной длины слова. Память содержала 4000 байт по 8 бит (в более поздних моделях объем увеличился до немыслимых в то время 16 000 байт).

Каждый байт содержал символ в 6 бит, административный бит и бит для обозначения конца слова. Команда `MOVE`, например, использовала два адреса: источника и приемника. Она перемещала байты из источника в приемник, пока не обнаруживала бит конца слова, установленный в 1.

В 1964 году маленькая, никому не известная компания CDC (Control Data Corporation) выпустила машину 6600, которая работала почти на порядок быстрее, чем 7094 (и все остальные машины того времени). Этот компьютер для сложных расчетов пользовался большой популярностью, и компания CDC пошла «в гору». Секрет столь высокого быстродействия заключался в том, что внутри ЦП (центрального процессора) находилась машина с высокой степенью параллелизма. У нее было несколько функциональных устройств для сложения, умножения и деления, и все они могли работать одновременно. Хотя быстрая работа машины требовала тщательной работы программиста, при определенных усилиях можно было сделать так, чтобы машина исполняла 10 команд одновременно.

Внутри машины 6600 было встроено несколько маленьких компьютеров. Таким образом, центральный процессор занимался только вычислениями, а остальные функции (управление работой машины, а также ввод и вывод информации) исполняли маленькие компьютеры. Можно сказать, что 6600 на десятилетия опередил свое время. Многие ключевые принципы современных компьютеров уходят корнями к 6600.

Разработчик компьютера 6600 Сеймур Крей (Seymour Cray) был легендарной личностью, как и фон Нейман. Он посвятил всю свою жизнь созданию очень мощных компьютеров, которые сейчас называют **суперкомпьютерами**. Среди них можно назвать 6600, 7600 и Cray-1. Сеймур Крей также является автором известного «алгоритма покупки автомобилей»: вы идете в магазин, ближайший к вашему дому, показываете на машину, ближайшую к двери, и говорите: «Я беру эту». Этот алгоритм позволяет тратить минимум времени на не очень важные дела (покупку автомобилей) и оставляет большую часть времени на важные (разработку суперкомпьютеров).

Следует упомянуть еще один компьютер — Burroughs B5000. Разработчики машин PDP-1, 7094 и 6600 занимались только аппаратным обеспечением, стараясь снизить его стоимость (DEC) или заставить работать быстрее (IBM и CDC). На программное обеспечение никто не обращал внимания. Производители B5000 пошли другим путем. Они разработали машину специально для программирования на языке Algol 60 (предшественнике языков C и Java), сконструировав аппаратное обеспечение так, чтобы упростить задачу компилятора. Так появилась идея, что при разработке компьютера нужно также учитывать и программное обеспечение. К сожалению, о ней практически сразу же забыли.

Третье поколение — интегральные схемы (1965–1980)

Изобретение кремниевой интегральной схемы в 1958 году Джеком Килби (Jack Kilby) и Робертом Нойсом (Robert Noyce) позволило разместить на одной небольшой микросхеме десятки транзисторов. Компьютеры на интегральных схемах были меньшего размера, работали быстрее и стоили дешевле, чем их предшественники на транзисторах.

К 1964 году компания IBM лидировала на компьютерном рынке, но существовала одна большая проблема: выпускаемые ей компьютеры 7094 и 1401, исключительно успешные и прибыльные, были несовместимы друг с другом. Один из них предназначался для сложных расчетов, в нем использовались параллельные двоичные операции с регистрами по 36 бит, во втором применялась десятичная система счисления и слова переменной длины. У многих покупателей были оба этих компьютера и им не нравилась необходимость содержать два разных отдела программирования, не имевшие ничего общего.

Когда пришло время заменить эти две серии компьютеров, компания IBM сделала решительный шаг. Она выпустила линейку транзисторных компьютеров System/360, которые были предназначены как для научных, так и для коммерческих расчетов. Линейка System/360 имела много нововведений. Это было целое семейство компьютеров для работы с одним языком (ассемблером), разного размера и вычислительной мощности. Компания смогла заменить 1401 на 360 (модель 30), а 7094 — на 360 (модель 75). Модель 75 была больше по размеру, работала быстрее и стоила дороже, но программы, написанные для одной из них, теоретически могли использоваться в другой. На практике программы, написанные для маленькой модели, исполнялись большой моделью без особых затруднений. Но в случае переноса программного обеспечения с большой машины на маленькую могло не хватить памяти. И все же создание такой линейки компьютеров было большим достижением. Идея создания семейств компьютеров вскоре стала очень популярной, и в течение нескольких лет большинство компьютерных компаний выпустили серии сходных машин с разной стоимостью и функциями. В табл. 1.2 показаны некоторые параметры первых моделей из семейства 360. О других моделях этого семейства мы расскажем далее.

Таблица 1.2. Первые модели серии IBM 360

Параметры	Модель 30	Модель 40	Модель 50	Модель 65
Относительная производительность	1	3,5	10	21
Время цикла, нс	1000	625	500	250
Максимальный объем памяти, байт	65 536	262 144	262 144	524 288
Количество байтов, вызываемых из памяти за один цикл	1	2	4	16
Максимальное количество каналов данных	3	3	4	6

Еще одно нововведение в 360 — **многозадачность**. В памяти компьютера могло находиться одновременно несколько программ, и пока одна программа ждала, когда закончится процесс ввода-вывода, другая исполнялась. В результате ресурсы процессора расходовались более рационально.

Компьютер 360 был первой машиной, которая могла полностью эмулировать (имитировать) работу других компьютеров. Младшие модели могли эмулировать 1401, а старшие — 7094, поэтому программисты могли оставлять свои старые программы без изменений и использовать их в работе с 360. Некоторые модели

360 исполняли программы, написанные для 1401, гораздо быстрее, чем сама 1401, поэтому заказчики вообще не переделывали свои программы.

Эмуляция на компьютерах серии 360 упрощалась тем, что исходные модели, как и большинство следующих моделей, использовали микропрограммы. Нужно было всего лишь написать три микропрограммы: одну — для системы команд 360, другую — для системы команд 1401, третью — для системы команд 7094. Эта гибкость стало одной из главных причин применения микропрограммирования. Сокращение количества электронных ламп в данном случае роли уже не играло, потому что в серии 360 их уже не было.

Компьютеру 360 удалось разрешить дилемму между двоичной и десятичной системами счисления: у этого компьютера было 16 регистров по 32 бит для двоичных операций, но память состояла из байтов, как у 1401. В 360 использовались такие же команды для перемещения записей переменного размера между блоками памяти, как и в 1401.

Другой особенностью 360 было громадное (на то время) адресное пространство 2^{24} байт (16 Мбайт). В те дни, когда байт памяти стоил несколько долларов, это казалось бесконечностью. К сожалению, линейка 360 позднее сменилась линейкой 370, затем 4300, 3080, 3090, и все эти компьютеры имели сходную архитектуру. К середине 80-х годов 16 Мбайт памяти стало недостаточно, и компании IBM пришлось частично отказаться от совместимости, чтобы перейти на 32-разрядную адресацию, необходимую для памяти объемом в 2^{32} байт.

Казалось бы, раз у машин были 32-разрядные слова и регистры, у них вполне могли бы быть и 32-разрядные адреса. Но в то время никто не мог даже представить себе компьютер с объемом памяти в 16 миллионов байт. Хотя переход на 32-разрядные адреса прошел удачно для IBM, он снова стал лишь временным решением проблемы адресации памяти, так как компьютерным системам вскоре потребуется адресовать более 2^{32} (4 294 967 296) байт памяти. В ближайшие несколько лет на сцене появятся компьютеры с 64-разрядным пространством памяти.

Мир мини-компьютеров сделал большой шаг вперед в третьем поколении вместе с производством линейки компьютеров PDP-11, 16-разрядных последователей PDP-8. Во многих отношениях компьютер PDP-11 был младшим братом 360, а PDP-1 — младшим братом 7094. И у 360, и у PDP-11 были регистры, связанные со словами, память с байтами, и в обеих линейках компьютеры имели разную стоимость и разные функции. PDP-1 широко использовался, особенно в университетах, и компания DEC продолжала лидировать среди производителей мини-компьютеров.

Четвертое поколение — сверхбольшие интегральные схемы (1980–?)

Появление **сверхбольших интегральных схем (СБИС)** в 80-х годах позволило размещать на одной плате сначала десятки тысяч, затем сотни тысяч и, наконец, миллионы транзисторов. Это привело к созданию компьютеров меньшего размера и более быстродействующих. До появления PDP-1 компьютеры были настолько большие и дорогостоящие, что компаниям и университетам приходилось иметь специальные отделы **(вычислительные центры)**. К 80-м годам цены упали так

сильно, что возможность приобретать компьютеры появилась не только у организаций, но и у отдельных людей. Началась эра персональных компьютеров.

Персональные компьютеры требовались совсем для других целей, чем их предшественники. Они применялись для обработки слов, электронных таблиц, а также для исполнения приложений с высоким уровнем интерактивности (например, игр), для которых большие компьютеры не подходили.

Первые персональные компьютеры продавались в виде комплектов. Каждый комплект содержал печатную плату, набор интегральных схем, обычно включая схему Intel 8080, несколько кабелей, источник питания и иногда 8-дюймовый дисковод. Сложить из этих частей компьютер покупатель должен был сам. Программное обеспечение к компьютеру не прилагалось. Покупателю приходилось писать программное обеспечение самому. Позднее появилась операционная система CP/M, написанная Гари Килдаллом (Gary Kildall) для Intel 8080. Это была полноценная операционная система (на дискете), со своей файловой системой и интерпретатором для исполнения пользовательских команд, которые вводились с клавиатуры.

Еще один персональный компьютер, Apple (а позднее и Apple II), был разработан Стивом Джобсом (Steve Jobs) и Стивом Возняком (Steve Wozniak). Этот компьютер стал чрезвычайно популярным среди домашних пользователей и школ, что в мгновение ока сделало компанию Apple серьезным игроком на рынке.

Наблюдая за тем, чем занимаются другие компании, компания IBM, лидирующая тогда на компьютерном рынке, тоже решила заняться производством персональных компьютеров. Но вместо того чтобы конструировать компьютер «с нуля» только из компонентов IBM (сделанных из транзисторов IBM, сделанных из кристаллов IBM), что заняло бы слишком много времени, компания сделала нечто неслыханное: она предоставила одному из своих работников, Филипу Эстриджу (Philip Estridge), большую сумму денег, приказала ему отправиться куда-нибудь подальше от вмешивающихся во все бюрократов главного управления компании, находящегося в Армонке (шт. Нью-Йорк). Эстридж, работая в 2000 километров от главного управления компании, взял за основу процессор Intel 8088 и построил персональный компьютер из разнородных компонентов. Этот компьютер (IBM PC) появился в 1981 году и стал самым покупаемым компьютером в истории. К 30-летию PC был опубликован ряд статей о его истории, включая статьи Брэдли [Bradley, 2011], Гота [Goth, 2011], Брайда [Bride, 2011] и Сингха [Singh, 2011].

Однако компания IBM сделала одну необычную вещь, о которой позже пожалела. Вместо того чтобы держать проект машины в секрете (или по крайней мере оградить себя огромной, непроницаемой стеной патентов), как она обычно делала, компания опубликовала полные проекты, включая все электронные схемы, в книге стоимостью 49 долларов. Эта книга была опубликована для того, чтобы другие компании могли производить сменные платы для IBM PC, что повысило бы совместимость и популярность этого компьютера. К несчастью для IBM, как только проект IBM PC стал широко известен, многие компании начали делать **клоны** PC и часто продавали их гораздо дешевле, чем IBM (поскольку все составные части компьютера можно было легко приобрести). Так началось бурное производство персональных компьютеров.

Хотя некоторые компании (такие как Commodore, Apple и Atari) производили персональные компьютеры с использованием своих процессоров, а не процессоров Intel, потенциал производства IBM PC был настолько велик, что другим компаниям приходилось пробиваться с трудом. Выжить удалось только некоторым из них, и то лишь потому, что они специализировались в узких областях, например в производстве рабочих станций или суперкомпьютеров.

Одной из таких выживших (хотя и с трудом) моделей стал компьютер Apple Macintosh. Он появился в 1984 году как наследник злополучного Apple Lisa — первого компьютера с графическим интерфейсом, сходным с интерфейсом популярной ныне системы Windows. Неудача Lisa объяснялась дороговизной, но более дешевый Macintosh, появившийся через год, пользовался огромным успехом и завоевал много преданных поклонников.

Ранний рынок персональных компьютеров также привел к неслыханному спросу на портативные компьютеры. В то время портативный компьютер казался такой же нелепицей, как и портативный холодильник. Первый портативный компьютер — Osborne-1 — весил 11 килограммов, и носить его с собой было, мягко говоря, неудобно. Тем не менее само появление этой модели доказало, что это возможно. Коммерческий успех Osborne-1 был довольно скромным, но через год фирма Compaq представила свой первый портативный клон IBM PC и быстро завоевала место лидера на рынке портативных компьютеров.

Первая версия IBM PC была оснащена операционной системой MS-DOS, которую выпускала тогда еще крошечная корпорация Microsoft. Благодаря тому, что фирма Intel выпускала все более мощные процессоры, IBM и Microsoft совместно разработали последовавшую за MS-DOS операционную систему OS/2, отличительной особенностью которой был **графический пользовательский интерфейс** (Graphical User Interface, GUI), сходный с интерфейсом Apple Macintosh. Между тем компания Microsoft также разработала собственную операционную систему Windows, которая работала на основе MS-DOS, — на случай, если OS/2 не будет иметь спроса. Короче говоря, OS/2 действительно не пользовалась спросом, а Microsoft успешно продолжала выпускать операционную систему Windows, что послужило причиной грандиозного раздора между IBM и Microsoft. История о том, как крошечная компания Intel и еще меньшая компания Microsoft умудрились свергнуть IBM, одну из самых крупных, самых богатых и самых влиятельных корпораций в мировой истории, подробно излагается в бизнес-школах всего мира.

Первоначальный успех процессора 8088 воодушевил компанию Intel на его дальнейшие усовершенствования. Особо примечательна версия 80386, выпущенная в 1985 году, — этот процессор был уже 32-разрядным. За ней последовал улучшенный вариант, который, естественно, назывался 80486. Последующие версии назывались Pentium и Core. Эти микросхемы используются практически во всех современных PC. Архитектура процессоров этого семейства часто обозначается общим термином **x86**. Совместимые микросхемы, производимые фирмой AMD, тоже называются x86.

В середине 80-х годов на смену CISC (Complex Instruction Set Computer — компьютер с полным набором команд) пришли компьютеры RISC (Reduced Instruction Set Computer — компьютер с сокращенным набором команд). RISC-команды были проще и выполнялись гораздо быстрее. В 90-х годах появились

суперскалярные процессоры, которые могли исполнять много команд одновременно, часто не в том порядке, в котором они располагаются в программе. Понятия RISC, CISC и суперскалярного процессора будут представлены в главе 2, где мы рассмотрим их более подробно.

Также в середине 1980-х годов Росс Фримен (Ross Freeman) со своими коллегами из Xilinx разработал изящный метод создания микросхем, для которого не требовались горы денег или доступ к производству кристаллического кремния. Новая разновидность микросхем, называемая **программируемой вентильной матрицей** (FPGA, Field-Programmable Gate Array), содержала большое количество универсальных логических элементов, которые можно было «запрограммировать» на любую функцию, необходимую устройству. Благодаря своему выдающемуся новому подходу к проектированию оборудование FPGA по своей гибкости не уступает программному обеспечению. Из FPGA стоимостью в несколько десятков или сотен долларов можно собрать компьютерную систему для уникальных приложений, с которыми работают всего несколько пользователей. К счастью, компании-производители кристаллического кремния продолжают миллионами выпускать более быстрые, дешевые и менее энергоемкие микросхемы для массового применения. Однако FPGA остаются популярным средством построения аппаратного обеспечения для узкого круга пользователей — например, построения прототипов, мелкосерийных приложений и образования.

Вплоть до 1992 года персональные компьютеры были 8-, 16- и 32-разрядными. Затем появилась революционная 64-разрядная модель Alpha производства DEC — самый что ни на есть настоящий RISC-компьютер, намного превзошедший по показателям производительности все прочие ПК. Впрочем, тогда коммерческий успех этой модели оказался весьма скромным — лишь через десятилетие 64-разрядные машины приобрели популярность, да и то лишь в качестве высокопроизводительных серверов.

В 1990-е годы компьютерные системы ускорялись посредством различных микроархитектурных оптимизаций, многие из которых будут рассмотрены в книге. Пользователи таких систем пребывали в благодушном настроении, потому что в каждой новой купленной системе их программы работали намного быстрее, чем в старой. Однако к концу 1990-х годов тенденция к повышению скорости стала снижаться из-за двух важных препятствий в области проектирования: архитекторы исчерпали запас возможностей для ускорения программ, а охлаждение процессоров стало обходиться слишком дорого. Многие компьютерные компании, отчаянно стремившиеся к построению более быстрых процессоров, обратились к параллельным архитектурам как к средству выжать больше быстродействия из своей электроники. В 2001 году фирма IBM представила двухъядерную архитектуру POWER4 — первый образец крупносерийного центрального процессора, включавшего два процессора на одной подложке. В наши дни большинство процессоров для настольных систем и серверов и даже некоторые встроенные процессоры состоят из нескольких процессоров. К сожалению, для рядового потребителя производительность таких мультипроцессоров была довольно скромной, потому что (как мы увидим в следующих главах) для эффективной работы параллельных машин программист должен явным образом организовать параллельное выполнение программ, а эта задача сложна и подвержена ошибкам.

Пятое поколение — компьютеры небольшой мощности и невидимые компьютеры

В 1981 году правительство Японии объявило о намерениях выделить национальным компаниям 500 миллионов долларов на разработку компьютеров пятого поколения на основе технологий искусственного интеллекта, которые должны были потеснить «послушные» машины четвертого поколения. Наблюдая за тем, как японские компании оперативно захватывают рыночные позиции в самых разных областях промышленности — от фотоаппаратов до стереосистем и телевизоров, — американские и европейские производители в панике бросились требовать у своих правительств аналогичных субсидий и прочей поддержки. Однако несмотря на большой шум, японский проект разработки компьютеров пятого поколения в конечном итоге показал свою несостоятельность и был тихо свернут. В каком-то смысле эта ситуация оказалась близка той, с которой столкнулся Беббидж — идея настолько опередила свое время, что для ее реализации не нашлось адекватной технологической базы.

Тем не менее, то, что можно назвать пятым поколением компьютеров, все же материализовалось, но в весьма неожиданном виде — компьютеры начали стремительно уменьшаться. В 1989 году фирма Grid Systems выпустила первый планшетный компьютер, который назывался GridPad. Он был оснащен небольшим экраном, на котором пользователь мог писать специальным пером. Такие системы, как GridPad, продемонстрировали, что компьютер не обязан стоять на столе или в серверной — пользователь может носить его с собой, а с сенсорным экраном и распознаванием рукописного текста он становится еще более удобным.

Модель Apple Newton, появившаяся в 1993 году, наглядно доказала, что компьютер можно уместить в корпусе размером с кассетный плеер. Как и GridPad, Newton использовал рукописный ввод, что на первых порах стало большим препятствием на пути к успеху. Но впоследствии пользовательский интерфейс подобных машин, которые теперь называются **персональными электронными секретарями** (Personal Digital Assistants, **PDA**), или просто **карманными компьютерами**, был усовершенствован и приобрел широкую популярность. В наши дни очередным этапом их эволюции стали **смартфоны**.

Интерфейс рукописного ввода PDA был усовершенствован Джеффом Хокинсом (Jeff Hawkins), создавшим компанию Palm для разработки деловых карманных компьютеров, рассчитанных на массового потребителя. Хокинс по образованию был инженером-электротехником, но он живо интересовался нейробиологией (наука о человеческом мозге). Он понял, что для повышения надежности рукописного ввода можно обучить пользователей приемам, которые упрощали восприятие ввода компьютером — технология, получившая название «Graffiti», требовала непродолжительного обучения пользователя, но в конечном итоге повышала скорость и надежность ввода. Первый карманный компьютер Palm — Palm Pilot — пользовался огромным успехом, а технология «Graffiti», ставшая одним из выдающихся достижений в компьютерной области, убедительно продемонстрировала возможности человеческого разума по использованию возможностей человеческого разума.

Пользователи PDA обожали свои устройства, прилежно используя их для управления своим расписанием и контактами. В 1990-е годы сотовые телефоны

получили широкое распространение. Фирма IBM встроила сотовый телефон в PDA, создав так называемый «смартфон». В первом смартфоне, который назывался Simon, для ввода использовался сенсорный экран, а в распоряжении пользователя оказывались все возможности PDA, а также телефон, игры и электронная почта. Уменьшение размеров и стоимости компонентов в конечном итоге привело к массовому распространению смартфонов. Сейчас наибольшей популярностью пользуются платформы Apple iPhone и Google Android.

Но даже карманные компьютеры не стали по-настоящему революционной разработкой. Значительно большее значение придается так называемым «невидимым» компьютерам — тем, что встраиваются в бытовую технику, часы, банковские карточки и огромное количество других устройств [Bechini et al., 2004]. Процессоры этого типа предусматривают широкие функциональные возможности и не менее широкий спектр вариантов применения за весьма умеренную цену. Вопрос о том, можно ли свести эти микросхемы в одно полноценное поколение (а существуют они с 1970-х годов), остается открытым. Факт в том, что они на порядок расширяют возможности тысяч бытовых и других устройств. Уже сейчас влияние невидимых компьютеров на развитие мировой промышленности очень велико, и с годами оно будет возрастать. Одной из особенностей такого рода компьютеров является то, что их аппаратное и программное обеспечение зачастую проектируется методом **соразработки** [Henkel et al., 2003], о котором мы поговорим далее в этой книге.

Итак, к первому поколению причисляются компьютеры на электронных лампах (такие, как ENIAC), ко второму — транзисторные машины (IBM 7094), к третьему — первые компьютеры на интегральных схемах (IBM 360), к четвертому — персональные компьютеры (линейки ЦП Intel). Что же касается пятого поколения, то оно больше ассоциируется не с конкретной архитектурой, а со сменой парадигмы. Компьютеры будущего будут встраиваться во все мыслимые и немыслимые устройства и за счет этого действительно станут невидимыми. Они прочно войдут в повседневную жизнь — будут открывать двери, включать лампы, распределять деньги и выполнять тысячи других обязанностей. Эта модель, разработанная Марком Вайзером (Mark Weiser) в поздний период его деятельности, первоначально получила название **повсеместной компьютеризации**, но в настоящее время не менее распространен термин **всепроникающая компьютеризация** [Weiser, 2002]. Это явление обещает изменить мир не менее радикально, чем промышленная революция. Мы не будем останавливаться на этом вопросе подробно, но если вам интересно, можете обратиться к дополнительной литературе [Lyytinen and Yoo, 2002; Saha and Mukherjee, 2003; Sakamura, 2002].

Типы компьютеров

В предыдущем разделе мы кратко изложили историю компьютерных систем. В этом разделе мы расскажем о положении дел в настоящий момент и сделаем некоторые предположения на будущее. Хотя персональные компьютеры — наиболее известные типы «умных» машин, в наши дни существуют и другие типы машин, поэтому стоит кратко рассказать о них.

Технологические и экономические аспекты

По темпам развития компьютерная промышленность опережает все остальные отрасли. Главная движущая сила — способность производителей помещать с каждым годом все больше и больше транзисторов на микросхему. Чем больше транзисторов (крошечных электронных переключателей), тем больше объем памяти и мощнее процессоры. Гордон Мур (Gordon Moore), один из основателей и бывший председатель совета директоров Intel, однажды сострил по поводу того, что если бы авиационные технологии развивались с такой же скоростью, как компьютерные, самолеты стоили бы 500 долларов и облетали землю за 20 минут на 20 литрах топлива. Правда, для этого они должны стать размером с обувную коробку.

Он же сформулировал закон технологического прогресса, известный теперь под именем **закона Мура**. Когда Гордон готовил доклад для одной из промышленных групп, он заметил, что каждое новое поколение микросхем появляется через три года после предыдущего. Поскольку у каждого нового поколения компьютеров было в 4 раза больше памяти, чем у предыдущего, стало понятно, что число транзисторов на микросхеме возрастает в постоянной пропорции, и таким образом, этот рост можно предсказать на годы вперед. Закон Мура часто представляется в формулировке, которая гласит, что число транзисторов на одной микросхеме удваивается каждые 18 месяцев, то есть увеличивается на 60 % каждый год. Размеры микросхем и даты их производства подтверждают, что закон Мура действует до сих пор (рис. 1.6).

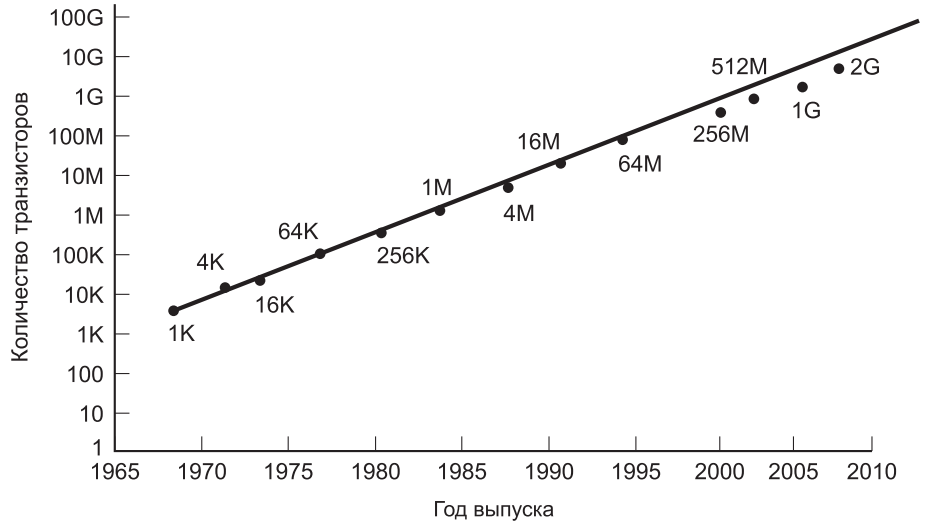


Рис. 1.6. Закон Мура предсказывает, что количество транзисторов на одной микросхеме увеличивается на 60 % каждый год. Точки на графике — объем памяти в битах

По большому счету, закон Мура — это никакой не закон, а простое эмпирическое наблюдение о том, с какой скоростью физики и инженеры-технологи развивают компьютерные технологии, и предсказание, что с такой скоростью они будут работать и в будущем. Многие специалисты считают, что закон Мура

будет действовать еще лет десять, а возможно, и дольше. Другие предсказывают, что довольно скоро разработчики столкнутся с рассеянием энергии, утечками тока и другими проблемами, которые придется каким-то образом решать [Bose, 2004; Kim et al., 2003]. Впрочем, уменьшение транзисторов скоро приведет к тому, что толщина этих устройств уменьшится до нескольких атомов. На этой стадии транзисторы станут слишком малыми для надежной работы или попросту достигнут точки, в которой дальнейшее уменьшение потребует субатомных структурных элементов. Несмотря на все будущие проблемы с законом Мура, на горизонте уже появились перспективные технологии — такие, как квантовые вычисления [Oskin et al., 2002] и углеродные нанотрубки [Heinze et al., 2002]. Возможно, они позволят вывести масштабы электронных компонентов за ограничения, присущие кремнию.

Закон Мура связан с тем, что некоторые экономисты называют **эффективным циклом**. Достижения в компьютерных технологиях (увеличение количества транзисторов на одной микросхеме) приводят к продукции лучшего качества и более низким ценам. Низкие цены ведут к появлению новых областей применения (никому не приходило в голову разрабатывать компьютерные игры, когда компьютер стоил 10 млн долларов — хотя когда цена снизилась до $120 000, студенты МТИ взялись за дело). Новые области применения приводят к возникновению новых компьютерных рынков и новых компаний. Существование всех этих компаний ведет к конкуренции между ними, которая, в свою очередь, порождает спрос на лучшие технологии. Круг замыкается.

Еще один фактор развития компьютерных технологий — первый закон программного обеспечения, названный в честь Натана Мирвольда (Nathan Myhrvold), одного из руководителей компании Microsoft. Этот закон гласит: «Программное обеспечение — это газ. Он распространяется и полностью заполняет резервуар, в котором находится». В 80-е годы электронная обработка текстов осуществлялась программами типа troff (именно программа troff использовалась при создании этой книги). Программа troff занимает несколько десятков килобайтов памяти. Современные электронные редакторы занимают десятки мегабайтов. В будущем, несомненно, они будут занимать десятки гигабайтов (в первом приближении приставки «кило», «мега» и «гига» означают «тысячу», «миллион» и «миллиард», соответственно; подробнее об этом см. раздел «Единицы измерения»). Программное обеспечение продолжает развиваться и порождает постоянный спрос на процессоры, работающие с более высокой скоростью, на память большего объема, на устройства ввода-вывода более высокой производительности.

С каждым годом количество транзисторов на одной микросхеме стремительно увеличивается, но темпы развития других компонентов компьютера столь же велики. Например, у машины IBM PC/XT, появившейся в 1982 году, объем жесткого диска составлял всего 10 Мбайт. Двадцать лет спустя в системах-наследниках PC/XT обычно устанавливаются жесткие диски емкостью 1 Тбайт. Разница на пять порядков стала возможной благодаря ежегодному приросту емкости в 50 %. Правда, подсчитать, насколько быстро происходит совершенствование жесткого диска, гораздо сложнее, поскольку тут есть несколько параметров (скорость передачи данных, время позиционирования, цена и т. д.), но измерение любого из этих параметров покажет, что с 1982 года соотношение «цена/производитель-

ность» возрастает по крайней мере на 50 % в год. Незаурядные достижения по части производительности дисков, равно как и то обстоятельство, что объем выручки от продажи дисков, выпущенных в Кремниевой долине, превысил аналогичный показатель по микросхемам процессоров, заставил Эла Хогланда (Al Hoagland) предположить, что это место следовало бы назвать Железооксидной долиной (ведь именно этот материал является носителем информации на дисках). Впрочем, сейчас положение дел медленно изменяется, так как во многих системах флэш-память на базе кремния начинает вытеснять традиционные вращающиеся диски.

Крупные достижения наблюдаются также и в сфере телекоммуникаций и создания сетей. Меньше чем за два десятилетия мы пришли от модемов, передающих информацию со скоростью 300 бит/с, к аналоговым модемам, работающим со скоростью 56 Кбит/с, и оптико-волоконным сетям, где скорость передачи уже больше 10^{12} бит/с. Оптико-волоконные трансатлантические телефонные кабели (например, TAT-12/13) стоят около 700 млн долларов, действуют в течение 10 лет и могут передавать 300 000 звонков одновременно, поэтому себестоимость 10-минутной межконтинентальной связи составляет менее одного цента. Лабораторные исследования подтвердили, что возможны системы связи, работающие со скоростью 1 Тбит/с (10^{12} бит/с) на расстоянии более 100 км без усилителей. Едва ли нужно упоминать здесь об экспоненциальном росте Интернета.

Широкий спектр компьютеров

Ричард Хемминг (Richard Hamming), бывший исследователь из Bell Laboratories, заметил, что количественное изменение величины на порядок ведет к качественному изменению. Например, гоночная машина, которая может ездить со скоростью 1000 км/ч по пустыне Невада, коренным образом отличается от обычной машины, которая ездит со скоростью 100 км/ч по шоссе. Точно так же небоскреб в 100 этажей несопоставим с десятиэтажным многоквартирным домом. А если речь идет о компьютерах, то тут за три десятилетия количественные показатели увеличились не в 10, а в 1 000 000 раз.

Развивать компьютерные технологии, исходя из закона Мура, можно двумя путями: создавать компьютеры все большей и большей мощности при постоянной цене или выпускать одну и ту же модель с каждым годом за меньшие деньги. Компьютерная промышленность идет по обоим путям, создавая широкий спектр разнообразных компьютеров. Очень приблизительная классификация современных компьютеров представлена в табл. 1.3.

В следующих разделах мы рассмотрим свойства, присущие каждой из вышеперечисленных категорий.

Одноразовые компьютеры

В самой верхней строчке находятся микросхемы, которые приклеиваются на внутреннюю сторону поздравительных открыток для проигрывания мелодий типа «Happy Birthday», свадебного марша или чего-нибудь в этом роде. Авторам пока не доводилось видеть на прилавках открытки с соболезнованиями, играющие похоронный марш, но, поскольку идея сформулирована, можно ожидать

Таблица 1.3. Типы современных компьютеров. Указанные цены приблизительны

Тип	Цена, долларов	Сфера применения
«Одноразовые» компьютеры	0,5	Поздравительные открытки
Встроенные компьютеры (микроконтроллеры)	5	Часы, машины, различные приборы
Мобильные и игровые компьютеры	50	Домашние компьютерные игры, смартфоны
Персональные компьютеры	500	Настольные и портативные компьютеры
Серверы	5000	Сетевые серверы
Мэйнфреймы	5 000 000	Пакетная обработка данных в банке

появления и таких открыток. Те, кто воспитывался на компьютерах стоимостью в миллионы долларов, воспринимают такие одноразовые компьютеры примерно так же, как одноразовый самолет.

Как бы то ни было, одноразовые компьютеры окружают нас. Вероятно, наиболее значимым достижением в этой области стало появление **микросхем RFID** (Radio Frequency Identification — радиочастотная идентификация). Теперь на безбатарейных микросхемах этого типа толщиной меньше 0,5 мм и себестоимостью в несколько центов устанавливаются крошечные приемопередатчики радиосигналов; кроме того, им присваивается уникальный 128-разрядный идентификатор. При получении импульса с внешней антенны они получают питание на время, достаточное для отправки ответного импульса со своим номером. Несмотря на крошечные размеры, спектр практического применения таких микросхем весьма значителен.

Взять хотя бы снятие штрих-кодов с товаров в магазинах. Уже проводились испытания, в ходе которых производители снабжали все выпускаемые ими товары микросхемами RFID (вместо штрих-кодов). При наличии таких микросхем покупатель может выбрать нужные продукты, положить их в корзину и, минуя кассу, выйти из магазина. По выходе считывающее устройство с антенной отсылает сигнал, заставляющий микросхемы на всех приобретенных товарах «рассказать» о себе, что они и делают путем беспроводной отсылки короткого импульса. Покупатель, в свою очередь, идентифицируется по микросхеме на его банковской/кредитной карточке. В конце каждого месяца магазин выставляет покупателю детализированный счет за все приобретенные за этот период товары. Если действующая банковская/кредитная карта на микросхеме RFID у покупателя не обнаруживается, звучит аварийный сигнал. Такая система позволяет не только избавиться от кассиров и очередей, но и защищает от краж — ведь прятать товары в карманах и сумках становится бессмысленно!

Между прочим, в отличие от штрих-кодов, которые идентифицируют только тип товара, 128-разрядные микросхемы RFID идентифицируют каждый конкрет-

ный экземпляр товара. Иными словами, каждая упаковка аспирина в супермаркете снабжается уникальным кодом RFID. Следовательно, если производитель аспирина обнаружит брак в одной из партий уже после ее появления в магазинах, он может оповестить об этом администрацию торговой сети, и каждый раз при покупке упаковки с идентификатором RFID, входящим в указанный «бракованный» диапазон, считывающее устройство будет генерировать звуковой сигнал. Для остальных упаковок тревога подниматься не будет.

Как бы то ни было, маркирование упаковок с аспирином, печенья и корма для животных — это лишь первый шаг в заданном направлении. Микросхемой ведь можно снабдить не только пакет собачьего корма, но и саму собаку! Уже сейчас многие собачники просят ветеринаров вживить в своих питомцев микросхемы RFID, чтобы в случае потери или кражи их можно было найти. Фермеры проделывают аналогичные операции с крупным рогатым скотом. Очевидно, на очереди — услуги по имплантации микросхем в детей не в меру боязливых родителей. Можно пойти и дальше, вживляя микросхемы в тела всех новорожденных — чтобы их, не дай бог, не перепутали! Правительства и органы внутренних дел, естественно, найдут тысячи достойных поводов к введению повсеместной «микросхематизации». В целом, все вышеупомянутые примеры, полагаю, достаточно ярко иллюстрируют возможности RFID.

Еще один полезный (хотя и более спорный) вариант применения микросхем RFID — это их установка на транспортных средствах. К примеру, если на каждом вагоне железнодорожного состава установлена микросхема, при его прохождении рядом со считывающим устройством подключенный к этому устройству компьютер может составлять список вагонов. Это позволит без труда отследить местонахождение каждого конкретного вагона, за счет чего поставщикам, заказчикам и администрации железной дороги станет существенно легче жить. Аналогичная схема применима и к большегрузным автомобилям. Водители легковых машин с помощью микросхем RFID могут платить за проезд по платным магистралям.

Технология RFID также предусматривает возможность применения в багажных системах. Недавно в аэропорту Хитроу (Лондон) прошло тестирование экспериментальной системы, снимающей большинство проблем, связанных с багажом пассажиров. Все сумки пассажиров, подписавшихся на эту услугу, снабжались микросхемами RFID, транспортировались по территории аэропорта отдельно от других и доставлялись напрямую в гостиницу. Среди других полезных применений RFID — определение цвета кузовов автомобилей перед их покраской в цехе, изучение миграции животных, указание температурного режима стирки предметов одежды и т. д. Микросхемы можно снабжать датчиками, в этом случае текущие показания температуры, давления, влажности и многие другие параметры окружающей среды сохраняются в младших разрядах.

Современные микросхемы RFID предусматривают возможность долговременного хранения. На этом основании Европейский Центробанк принял решение наладить в ближайшие годы выпуск банкнот с вживленными микросхемами. Такие банкноты будут запоминать все «инстанции», через которые они прошли. Подобным способом предполагается решить сразу несколько проблем, в частности, осложнить жизнь фальшивомонетчикам, отслеживать место получения выкупов при похищении людей и движение денег, полученных в результате ограблений, а также усилить меры противодействия отмыванию денег с возможно-

стью признания недействительными участвующих в подобных операциях купюр. Поскольку деньги с микросхемами потеряют такое свойство, как анонимность, полиции будет проще отслеживать преступников по движению купюр, которыми они пользуются. В конце концов, зачем вживлять микросхемы в людей, если ими переполнены их кошельки? Правда, когда общественность в полной мере осознает возможности технологии RFID, на эту тему следует ожидать бурных дискуссий.

Технологическая основа RFID стремительно развивается. Наиболее миниатюрные из микросхем этого типа пассивны (не содержат внутреннего источника питания), а их возможности ограничиваются передачей уникальных идентификаторов по внешним запросам. Более крупные микросхемы RFID активны, в них могут быть встроены аккумуляторы и элементарный компьютер, а соответственно, они способны выполнять определенный набор вычислительных операций. В эту последнюю категорию, помимо прочих, входят смарт-карты, применяемые в финансовых операциях.

Активность/пассивность микросхем RFID не является единственным параметром их классификации. Такие микросхемы также различаются по применяемым радиочастотным диапазонам. Чем ниже частота, тем ниже скорость передачи данных, но в то же время тем больше расстояние от антенны, на котором возможно считывание информации с микросхемы. Соответственно, микросхемы, работающие на высоких частотах, характеризуются высокой скоростью передачи данных и весьма ограниченным радиусом действия. Технология RFID постоянно совершенствуется, и если вас интересует эта тема, в Интернете можно найти массу информации по ней. Начать рекомендуем с сайта www.rfid.org.

Микроконтроллеры

Вторую строку в таблице занимают компьютеры, которыми оснащаются разного рода бытовые устройства. Такого рода встроенные компьютеры, называемые также **микроконтроллерами**, выполняют функцию управления устройствами и организации их пользовательских интерфейсов. Диапазон устройств, работающих с помощью микрокомпьютеров, крайне широк (примеры даются в скобках):

- бытовые приборы (будильники, стиральные машины, сушильные аппараты, микроволновые печи, охранные сигнализации);
- коммуникаторы (беспроводные и сотовые телефоны, факсимильные аппараты, пейджеры);
- периферийные устройства (принтеры, сканеры, модемы, приводы CD-ROM);
- развлекательные устройства (видеомагнитофоны, DVD-плееры, музыкальные центры, MP3-плееры, телеприставки);
- формирователи изображений (телевизоры, цифровые фотокамеры, видеокамеры, объективы, фотокопировальные устройства);
- медицинское оборудование (рентгеноскопические аппараты, томографы, кардиомониторы, цифровые термометры);
- военные комплексы вооружений (крылатые ракеты, межконтинентальные баллистические ракеты, торпеды);

- торговое оборудование (торговые автоматы, кассовые аппараты);
- игрушки (говорящие куклы, приставки для видеоигр, радиоуправляемые машинки и лодки).

В любой современной машине представительского класса устанавливается по полсотни микроконтроллеров, которые управляют различными подсистемами, в частности автоблокировкой колес, впрыском топлива, магнитолой, освещением и системой навигации. В реактивных самолетах количество микроконтроллеров достигает 200 и даже больше! В любом домашнем хозяйстве имеется по несколько сот компьютеров, причем члены семьи зачастую даже не подозревают об их существовании. Через несколько лет практически все приборы, работающие на источниках электропитания, будут оснащаться микроконтроллерами. По объемам ежегодных продаж микроконтроллеры опережают компьютеры всех остальных типов (за исключением одноразовых) на несколько порядков.

В отличие от микросхем RFID, выполняющих минимальный набор функций, микроконтроллеры хоть и невелики по размерам, но представляют собой полноценные вычислительные устройства. Каждый микроконтроллер состоит из процессора, памяти и средств ввода-вывода. Ввод-вывод, как правило, осуществляется отслеживанием состояния кнопок и переключателей с контролем состояния световых индикаторов, дисплея и звуковых компонентов устройства. Программное обеспечение микроконтроллеров в большинстве случаев «прошивается» производителем в виде постоянной памяти. Все микроконтроллеры можно разделить на два типа: универсальные и специализированные. Первые фактически являют собой обычные компьютеры, уменьшенные в размере. Специализированные же микроконтроллеры отличаются индивидуальной архитектурой и набором команд, приспособленными для решения определенного круга задач, например, связанных с воспроизведением мультимедийных данных. Микроконтроллеры бывают 4-, 8-, 16- и 32-разрядными.

Как бы то ни было, даже между универсальными микроконтроллерами, с одной стороны, и стандартными ПК, с другой, наблюдаются существенные различия. Во-первых, спрос на микроконтроллеры в максимальной степени обусловлен ценами на них. Принимая решение о закупке миллионной партии таких устройств, крупный заказчик может выбрать другого производителя, если тот предложит цену на один цент (за штуку) меньше, чем конкуренты. Поэтому, разрабатывая архитектуру для микроконтроллеров, производители всеми силами стараются оптимизировать производственные издержки, не слишком задумываясь о расширении функций. Цены на микроконтроллеры определяются разрядностью, типом, емкостью памяти и рядом других факторов; для сведения отметим, что при оптовых закупках 8-разрядных микроконтроллеров цена за штуку падает до 10 центов. Именно цена позволяет устанавливать микроконтроллеры в будильники за 10 долларов.

Во-вторых, почти все микроконтроллеры работают в реальном времени. За каждым входным сигналом должен следовать незамедлительный отклик. К примеру, после нажатия пользователем кнопки во многих приборах включается световой индикатор, причем между первым и вторым событием не должно быть никаких пауз. Необходимость работы в реальном времени зачастую определяет архитектурное решение микроконтроллеров.

В-третьих, встроенные системы зачастую ограничены по многим электрическим и механическим параметрам, таким как размер, вес и энергопотребление. С учетом этих ограничений и разрабатываются микроконтроллеры, устанавливаемые в такого рода системах.

Среди особенно интересных применений микроконтроллеров можно выделить встроенную управляющую платформу Arduino, разработанную Массимо Банци (Massimo Banzi) и Дэвидом Куартиллье (David Cuartielles) в Иврее (Италия). Целью данного проекта было создание полноценной встроенной управляющей платформы, которая бы стоила меньше большой пиццы, чтобы быть доступной для студентов и любителей электроники. (Задача была трудной — в Италии пицца очень популярна и поэтому стоит дешево.) Проектировщики хорошо справились со своей задачей; полноценная система Arduino стоит менее 20 долларов!

Аппаратная структура Arduino распространяется открыто; это означает, что вся информация опубликована и находится в свободном доступе, чтобы любой желающий мог строить (и даже продавать) системы на базе Arduino. Она базируется на 8-разрядном RISC-микроконтроллере Atmel AVR, причем большинство плат также включает базовую поддержку ввода-вывода. Плата программируется на языке программирования встроенных систем Wiring, который содержит все необходимое для управления устройствами реального времени. С платформой Arduino интересно работать прежде всего из-за большого и активного сообщества разработчиков. Опубликованы тысячи проектов, использующих Arduino — от электронных детекторов загрязнения окружающей среды до байкерской куртки с поворотными сигналами, от датчика влажности, отправляющего электронную почту при необходимости поливки растений, до беспилотных летательных аппаратов. Чтобы больше узнать об Arduino и заняться практическим созданием собственных проектов Arduino, обращайтесь на сайт www.arduino.cc.

Мобильные и игровые компьютеры

К следующей категории относятся мобильные и игровые компьютеры. В сущности, это обычные компьютеры, в которых расширенные возможности графических и звуковых контроллеров сочетаются с ограничениями по объему ПО и пониженной расширяемостью. Первоначально в эту категорию входили компьютеры с процессорами низших моделей для простых телефонов и игр типа пинг-понга, которые предусматривали вывод изображения на экран телевизора. С годами они превратились в достаточно мощные системы, которые по некоторым параметрам производительности ничем не хуже, а иногда даже лучше персональных компьютеров.

Чтобы получить представление о том, чем комплектуются игровые компьютеры, рассмотрим конфигурации трех популярных моделей этой категории. Первая из них — Sony PlayStation 3. В ней установлен многоядерный специализированный процессор с частотой 3,2 ГГц (он называется Cell) на базе RISC-процессора IBM PowerPC, и семь 128-разрядных элементов SPE (Synergistic Processing Elements). PlayStation 3 также оснащается 512 Мбайт памяти, графическим процессором Nvidia с частотой 550 МГц и проигрывателем Blu-ray. Вторая модель — Microsoft Xbox 360 — содержит трехъядерный процессор IBM PowerPC с частотой 3,2 ГГц с 512 Мбайт памяти, графическим процессором ATI с частотой

500 МГц, DVD-проигрывателем и жестким диском. Третья модель — планшет Samsung Galaxy — использовалась для вычитки исходного варианта этой книги. Она содержит двухъядерный ARM-процессор с частотой 1 ГГц, а также графический процессор (интегрированный в однокристальную систему Nvidia Tegra 2), 1 Гбайт памяти, две камеры, 3-осевой гиродатчик и флэш-память.

Возможно, все эти компьютеры не дотягивают по производительности до высокопроизводительных ПК, выпущенных в тот же период времени, но и отстают от них не сильно. Более того, некоторые компоненты даже мощнее своих аналогов, применяемых в ПК, — взять хотя бы 128-разрядные элементы SPE PlayStation 3 (по разрядности они превосходят все существующие модели ПК). Основное различие между этими машинами и ПК, впрочем, состоит не в производительности процессора, а в том, что игровые компьютеры представляют собой закрытые, законченные системы. Расширяемость таких систем при помощи сменных плат не предусмотрена, хотя в некоторых моделях присутствуют интерфейсы USB и FireWire. Что еще важнее, игровые компьютеры оптимизированы для конкретной области применения — трехмерных игр с высоким уровнем интерактивности и мультимедийным выводом. Все остальные функции считаются вторичными. Ограничения по части аппаратного и программного обеспечения, низкие тактовые частоты, недостаточный объем памяти, отсутствие монитора с высоким разрешением и (как правило) жесткого диска — все это позволяет продавать игровые системы по более низким ценам, чем персональные компьютеры. И действительно, несмотря на упомянутые ограничения, игровые компьютеры продаются миллионами, а их популярность только растет.

У мобильных компьютеров появляется дополнительное ограничение: они должны потреблять как можно меньше энергии для решения своих задач. Чем меньше энергии они потребляют, тем дольше проработают их батареи. Это требование создает немало проблем проектировщикам, потому что мобильные платформы (такие, как планшеты и смартфоны) должны бережно расходовать энергию, но при этом пользователи ожидают от них поддержки высокопроизводительных функций — трехмерной графики, обработки мультимедийных данных в высоком разрешении и качественных игр.

Персональные компьютеры

В следующую категорию входят персональные компьютеры. Именно они ассоциируются у большинства людей со словом «компьютер». Персональные компьютеры делятся на две основных категории: настольные и портативные (ноутбуки). Как правило, те и другие комплектуются модулями памяти общей емкостью в несколько гигабайт, жестким диском с данными на несколько терабайтов, приводом CD-ROM/DVD/Blu-ray, звуковой картой, сетевым интерфейсом, монитором с высоким разрешением и другими периферийными устройствами. На них устанавливаются сложные операционные системы, они расширяемы, при работе с ними используется широкий спектр программного обеспечения.

Центральным компонентом любого персонального компьютера является печатная плата, на которой устанавливаются процессор, память и устройства ввода-вывода (звуковая плата, возможно — модем и т. д.), а также интерфейсы клавиатуры, мыши, дискового привода, сетевой платы и прочих периферийных устройств, а также расширительные гнезда. Одна из таких плат изображена на рис. 1.7.

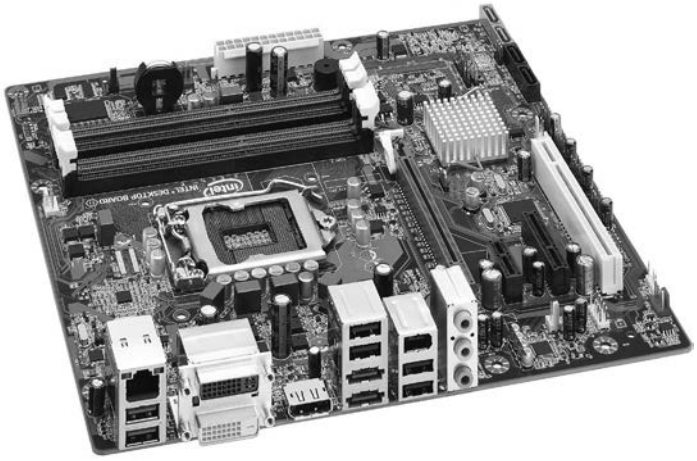


Рис. 1.7. Центральным компонентом любого персонального компьютера является печатная плата. На рисунке изображена системная плата Intel DQ67SW (фотография используется с разрешения корпорации Intel)

Ноутбуки, кроме своей компактности, ничем не отличаются от настольных ПК. В них устанавливаются аналогичные, хотя и меньшие по размеру, аппаратные компоненты. По возможностям исполнения и набору программ настольные и портативные компьютеры не различаются. Большинство читателей, вероятно, знакомы с характеристиками персональных компьютеров, поэтому не будем углубляться в анализ этой категории.

Еще одна вариация на эту тему — планшетные компьютеры (как, например, популярные iPad). Эти устройства представляют собой обычный PC в компактном исполнении, твердотельным накопителем вместо традиционного жесткого диска, сенсорным экраном и процессором, отличным от традиционного x86. Но с точки зрения архитектуры планшет представляет собой ноутбук с другим форм-фактором.

Серверы

Мощные персональные компьютеры и рабочие станции часто используются в качестве сетевых серверов — как в локальных сетях (обычно в пределах одной организации), так и в Интернете. Серверы, как правило, поставляются в однопроцессорной и мультипроцессорной конфигурациях. В системах из этой категории обычно устанавливаются модули памяти общим объемом в несколько гигабайтов, жесткие диски емкостью в терабайты и высокоскоростные сетевые интерфейсы. Некоторые серверы способны обрабатывать тысячи транзакций в секунду.

С точки зрения архитектуры, однопроцессорный сервер не слишком отличается от персонального компьютера. Он просто работает быстрее, занимает больше места, содержит больше дискового пространства и устанавливает более скоростные сетевые соединения. Серверы работают под управлением тех же операционных систем, что и персональные компьютеры, как правило, это различные версии UNIX и Windows.

Кластеры

В связи с тем, что по соотношению «цена/производительность» позиции рабочих станций и персональных компьютеров постоянно улучшаются, в последние годы появилась практика их объединения в **кластеры**. Кластер состоит из нескольких стандартных серверных систем, подключенных друг к другу по высокоскоростной сети и снабженных специальным программным обеспечением, которое позволяет направлять их ресурсы на решение единых задач (как правило, научных и инженерных). В большинстве случаев компоненты кластера — это совершенно обычные коммерческие машины, которые можно приобрести по отдельности в любом компьютерном магазине. Основным дополнением становятся высокоскоростные сетевые соединения, которые, как правило, тоже можно организовать при помощи стандартных сетевых плат.

Большие кластеры обычно размещаются в специальных залах или зданиях, называемых **центрами обработки данных**. Размеры центров обработки данных изменяются в широких пределах, от десятков до сотен тысяч и более машин. Количество компонентов кластера обычно ограничивается лишь толщиной кошелька покупателя. Поскольку компоненты кластеров достаточно дешевы, их приобретение для внутреннего использования могут себе позволить даже небольшие организации. Термины «кластер» и «центр обработки данных» часто используются как синонимы, хотя формально первое — совокупность серверов, а второе — зал или здание.

Нередко кластеры используются для создания веб-серверов. Если частота обращений к страницам веб-сайта исчисляется тысячами в секунду, самым экономичным решением обычно оказывается организация кластера из нескольких сотен (или даже тысяч) серверов и распределение между ними нагрузки по обработке запросов. Например, у Google по всему миру размещены центры обработки данных для обслуживания поисковых запросов; самый большой центр в Далласе (штат Орегон) занимает площадь двух футбольных полей. Место было выбрано из-за того, что центры обработки данных потребляют огромное количество электроэнергии, а Далласе находится рядом с 2-гигаваттной гидроэлектростанцией, которая может эту энергию поставлять. В общей сложности в центрах обработки данных Google установлено более 1 000 000 серверов.

Компьютерная отрасль динамично развивается, в ней постоянно что-то происходит. В 1960-е годы в ней преобладали гигантские компьютеры-мэйнфреймы (см. ниже), стоившие десятки миллионов долларов. Пользователи подключались к таким компьютерам с маленьких удаленных терминалов. Эта модель вычислений обладала высокой централизацией. Затем в 1980-е годы на сцене появились персональные компьютеры. Миллионы людей купили себе их, и вычисления стали децентрализованными.

С появлением центров обработки данных мы отчасти возвращаемся к прошлому в форме **облачных технологий** — своего рода «мэйнфреймам версии 2.0». Идея заключается в том, что у каждого пользователя имеется несколько простых устройств: PC, ноутбуков, планшетов и смартфонов, которые, по сути, предоставляют пользовательский интерфейс к облаку (то есть центрам обработки данных), в котором хранятся все фотографии, видеоролики, музыка и другие данные пользователя. В такой модели данные доступны пользователю в любой момент и на

любом устройстве и ему не нужно помнить, что где хранится. Центр обработки данных заменяет один большой централизованный компьютер, но мы возвращаемся к прежней парадигме: пользователи работают за простыми терминалами, а данные и вычислительные мощности находятся где-то в другом месте.

Кто знает, как долго проживет эта модель? Вполне возможно, что в ближайшие 10 лет так много людей начнет хранить свою музыку, фотографии и видеоролики в облаке, что (беспроводная) инфраструктура для взаимодействия с ним будет полностью парализована. Это может привести к новой революции: персональные компьютеры и локальное хранение данных на машинах пользователей, чтобы избежать сетевых «заторов».

Мэйнфреймы

Наконец мы дошли до больших компьютеров размером с комнату, напоминающих компьютеры 60-х годов и традиционно называемых мэйнфреймами. В большинстве случаев эти системы — прямые потомки больших компьютеров серии 360. Обычно они работают не намного быстрее, чем мощные серверы, но у них выше скорость процессов ввода-вывода и они часто оснащаются огромными дисковыми массивами, в которых хранятся многие тысячи гигабайт информации. Такие системы обходятся дорого, но часто продолжают работать из-за значительных вложений в программное обеспечение, данные и персонал, обслуживающий эти компьютеры. Многие компании считают, что дешевле заплатить несколько миллионов долларов один раз за такую систему, чем даже думать о необходимости заново переписывать все приложения для меньших компьютеров.

Именно этот класс компьютеров привел к проблеме 2000 года. Проблема возникла из-за того, что в 60-е и 70-е годы программисты, писавшие программы на языке COBOL, для экономии памяти представляли год двузначным десятичным числом. Они не смогли предвидеть, что их программное обеспечение будет использоваться через три или четыре десятилетия. Катастрофы, о которой так много говорили, не произошло, поскольку на ее предотвращение были затрачены огромные ресурсы, однако многие компании повторили ту же ошибку, добавив к числу года всего два десятичных разряда. Авторы этой книги предсказывают, что конец цивилизации произойдет в полночь 31 декабря 9999 года, когда сразу уничтожатся все программы, написанные за 8000 лет на языке COBOL.

Кроме выполнения программ, накопленных за последние 40 лет, в последние годы мэйнфреймы начали возрождаться под влиянием Интернета. Они заняли нишу мощных серверов Интернета, способных обрабатывать огромное количество транзакций в секунду, что крайне актуально для электронной коммерции в целом, и компаний, вынужденных обслуживать громадные базы данных в частности.

До последнего времени существовала еще одна крупная категория вычислительных машин — **суперкомпьютеры**. Их процессоры работали с очень высокой скоростью, в них устанавливались модули памяти общей емкостью в несколько десятков гигабайтов, высокоскоростные диски и сетевые интерфейсы. Суперкомпьютеры используются для решения различных научных и технических задач, которые требуют сложных вычислений, например таких, как моделирование сталкивающихся галактик, синтез новых лекарственных препаратов, моделирование потока воздуха вокруг крыла самолета. Сейчас, когда вычислительные

возможности, аналогичные тем, что предлагают суперкомпьютеры, реализуются в виде кластеров, эта категория компьютеров постепенно отмирает.

Семейства компьютеров

Основное внимание в этой книге уделяется трем популярным архитектурам наборов команд (ISA): x86, ARM и AVR. Архитектура x86 встречается практически во всех персональных компьютерах (PC с Windows и Linux, а также Mac) и серверных системах. Персональные компьютеры представляют интерес хотя бы по той причине, что все читатели ими, несомненно, пользуются. На серверных системах работают все интернет-сервисы. Архитектура ARM доминирует на мобильном рынке — большинство смартфонов и планшетных компьютеров использует процессоры ARM. Наконец, архитектура AVR задействована в дешевых микроконтроллерах, встречающихся во многих встроенных системах. Встроенные компьютеры, хотя и незаметны для пользователей, контролируют работу многих агрегатов в автомобилях, телевизорах, микроволновых печах, стиральных машинах, да и вообще практически во всех мыслимых электронных устройствах стоимостью выше 50 долларов. В этом разделе мы вкратце рассмотрим три архитектуры, которые далее по ходу изложения материала будем привлекать в качестве примеров.

Введение в архитектуру x86

В 1968 году Роберт Нойс (Robert Noyce), изобретатель кремниевой интегральной схемы, Гордон Мур (Gordon Moore), автор известного закона Мура, и Артур Рок (Arthur Rock), венчурный капиталист из Сан-Франциско, основали корпорацию Intel для производства компьютерных микросхем. За первый год своего существования корпорация продала микросхем всего на 3000 долларов, но потом объем продаж заметно вырос (в настоящее время Intel является крупнейшим мировым производителем процессоров).

В конце 60-х годов калькуляторы представляли собой большие электромеханические машины размером с современный лазерный принтер и весили около 20 кг. В сентябре 1969 года японская компания Busicom обратилась к корпорации Intel с просьбой выпустить 12 несерийных микросхем для электронной вычислительной машины. Инженер компании Intel Тед Хофф (Ted Hoff), назначенный в качестве исполнителя этого проекта, решил, что можно поместить 4-разрядный универсальный процессор на одну микросхему, которая будет выполнять те же функции и при этом окажется проще и дешевле. Так в 1970 году появился первый процессор на одной микросхеме — 4004 на 2300 транзисторах [Faggin et al., 1996].

Заметим, что ни сотрудники Intel, ни сотрудники Busicom не имели ни малейшего понятия, какое грандиозное открытие они совершили. Когда компания Intel решила, что стоит попробовать использовать процессор 4004 в других разработках, она предложила купить все права на новую микросхему у компании Busicom за 60 000 долларов, то есть за сумму, которую Busicom заплатила Intel за разработку этой микросхемы. Фирма Busicom сразу приняла предложение Intel, и компания Intel начала работу над 8-разрядной версией микросхемы, 8008, вы-

пущенной в 1972 году. Все процессоры семейства Intel, начиная с моделей 4004 и 8008, перечислены в табл. 1.4.

Таблица 1.4. Семейство процессоров Intel. Тактовая частота измеряется в МГц (1 МГц = 1 млн циклов/с)

Микросхема	Дата выпуска	МГц	Количество транзисторов	Объем памяти	Примечание
4004	4/1971	0,108	2 300	640 байт	Первый микропроцессор на микросхеме
8008	4/1972	0,08	3 500	16 Кбайт	Первый 8-разрядный микропроцессор
8080	4/1974	2	6 000	64 Кбайт	Первый многоцелевой процессор на микросхеме
8086	6/1978	5–10	29 000	1 Мбайт	Первый 16-разрядный процессор на микросхеме
8088	6/1979	5–8	29 000	1 Мбайт	Использовался в IBM PC
80286	2/1982	8–12	134 000	16 Мбайт	Появилась защита памяти
80386	10/1985	16–33	275 000	4 Гбайт	Первый 32-разрядный процессор
80486	4/1989	25–100	1 200 000	4 Гбайт	Кэш-память на 8 Кбайт
Pentium	3/1993	60–223	3 100 000	4 Гбайт	Два конвейера, у более поздних моделей — MMX
Pentium Pro	3/1995	150–200	5 500 000	4 Гбайт[1]	Два уровня кэш-памяти
Pentium II	5/1997	233–400	7 500 000	4 Гбайт	Pentium Pro плюс MMX
Pentium III	2/1999	650–1400	9 500 000	4 Гбайт	Появились SSE-команды, ускоряющие обработку трехмерной графики
Pentium 4	11/2000	1300–3800	42 000 000	4 Гбайт	Гиперпоточность, дополнительные SSE-команды
Core Duo	1/2006	1600–3200	152 000 000	2 Гбайт	Два ядра на одной подложке
Core	7/2006	1200–3200	410 000 000	64 Гбайт	64-разрядная 4-ядерная архитектура
Core i7	1/2011	1100–3300	1 160 000 000	24 Гбайт	Интегрированный графический процессор

[1] Шина адреса у микропроцессоров Pentium Pro и Pentium II имеет ширину 36 бит, что позволяет непосредственно адресовать 64 Гбайт. — *Примеч. науч. ред.*

Поскольку никто не ожидал большого спроса на микросхему 8008, она была выпущена достаточно ограниченным тиражом. Ко всеобщему удивлению, новая микросхема вызвала большой интерес, поэтому компания Intel начала разработку еще одного процессора, в котором предел в 16 Кбайт памяти (как у процессора 8008), навязываемый количеством внешних выводов микросхемы, был преодолен. Так появился небольшой универсальный процессор 8080, выпущенный в 1974 году. Как и PDP-8, он произвел революцию на компьютерном рынке и сразу стал массовым продуктом. Разница лишь в масштабах: компания DEC продала тысячи PDP-8, а Intel — миллионы процессоров 8080.

В 1978 году появился процессор 8086, 16-разрядный процессор на одной микросхеме. Процессор 8086 был во многом похож на 8080, но не был полностью совместим с ним. Затем появился процессор 8088 с такой же архитектурой, как у 8086. Он исполнял те же программы, что и 8086, но вместо 16-разрядной шины у него была 8-разрядная, из-за чего процессор работал медленнее, но стоил дешевле, чем 8086[1]. Когда компания IBM выбрала процессор 8088 для IBM PC, эта микросхема быстро превратилась в промышленный стандарт в области персональных компьютеров.

Ни 8088, ни 8086 не могли адресовать память объемом более 1 Мбайт. К началу 80-х годов это стало серьезной проблемой, поэтому компания Intel разработала модель 80286, совместимую с 8086. Основной набор команд остался в сущности таким же, как у процессоров 8086 и 8088, но организация памяти была несколько иной — и довольно неудобной из-за требования совместимости с предыдущими микросхемами и могла работать по-прежнему. Процессор 80286 использовался в IBM PC/AT и в моделях PS/2. Он, как и 8088, пользовался большим спросом (главным образом потому, что покупатели рассматривали его как более быстрый вариант модели 8088).

Следующим шагом был 32-разрядный процессор 80386, выпущенный в 1985 году. Как и 80286, он был более или менее совместим со всеми старыми версиями. Совместимость такого рода оказывалась благом для тех, кто пользовался старым программным обеспечением, и некоторым неудобством для тех, кто предпочитал современную архитектуру, не обремененную ошибками и технологиями прошлого.

Через четыре года появился процессор 80486. Он работал быстрее, чем 80386, мог исполнять операции с плавающей точкой и имел кэш-память объемом 8 Кбайт. **Кэш-память** позволяет держать наиболее часто используемые слова внутри центрального процессора и избегать (медленных) обращений к основной памяти. Процессор 80486 содержал встроенную поддержку мультипроцессорного режима, что давало производителям возможность конструировать системы с несколькими процессорами.

В этот момент компания Intel, проиграв судебную тяжбу по поводу нарушения правил именования товаров, узнала, что числа (например, 80486) не могут использоваться в качестве товарных знаков, поэтому следующее поколение компьютеров получило название Pentium (от греческого слова πεντε — пять). В от-

[1] На самом деле разница в стоимости самих микропроцессоров была незначительной, но компьютеры, собираемые на базе микропроцессора 8088, были дешевле, чем собираемые на базе микропроцессора 8086. В то время были распространены 8-разрядные периферийные устройства, поэтому микропроцессор 8088 позволял упростить сопряжение с внешними устройствами. — *Примеч. науч. ред.*

личие от процессора 80486, у которого был один внутренний конвейер, Pentium имел два, что позволяло работать ему почти в два раза быстрее (конвейеры мы рассмотрим подробно в главе 2).

Впоследствии в линейку Pentium были введены дополнительные команды, известные под общим названием **MMX** (MultiMedia eXtension — мультимедийное расширение). Они были предназначены для ускорения вычислительных операций, связанных с обработкой звуковых и видеоданных, что позволило отказаться от специальных мультимедийных сопроцессоров.

Когда появилось следующее поколение компьютеров, те, кто рассчитывал на название Sexium (sex по латыни — шесть), были разочарованы. Название Pentium стало так хорошо известно, что его решили оставить, и новую микросхему назвали Pentium Pro. Несмотря на столь незначительное изменение названия, этот процессор очень сильно отличался от предыдущего. У него была совершенно другая внутренняя организация и он мог исполнять до пяти команд одновременно.

Еще одно нововведение у Pentium Pro — двухуровневая кэш-память. Процессор содержал 8 Кбайт памяти для часто используемых команд и еще 8 Кбайт для часто используемых данных. В корпусе Pentium Pro рядом с процессором (но не на самой микросхеме) находилась другая кэш-память объемом в 256 Кбайт.

Большой объем кэш-памяти в Pentium Pro отчасти компенсировался отсутствием MMX-команд (первоначально Intel не удалось спроектировать микросхему адекватного размера, отвечавшую критерию рентабельности). Когда технологическая база позволила совместить в рамках одной микросхемы набор MMX-команд и большой кэш, новая модель получила название Pentium II. Через некоторое время для улучшенной передачи трехмерной графики в процессор были введены дополнительные мультимедийные команды под названием **SSE** (Streaming SIMD Extensions — потоковые SIMD-расширения) — в результате появился процессор Pentium III [Raman et al., 2000]. Правда, согласно внутренней номенклатуре компании это все тот же Pentium II.

Следующая модель Pentium получила новую внутреннюю архитектуру. Одновременно было решено перейти с римских цифр в обозначениях моделей на арабские. Так появился процессор Pentium 4. По традиции он превосходил все предыдущие модели по быстродействию. В версии с тактовой частотой 3,06 ГГц была реализована новая функция — гиперпоточность (hyperthreading). Она позволяет программам разделять задачи на два программных потока, которые обрабатываются процессором параллельно; следовательно, скорость исполнения повышается. Кроме того, для дальнейшего повышения скорости обработки звуковых и видеоданных был внедрен дополнительный набор SSE-команд.

В 2006 году фирма Intel сменила название бренда Pentium на Core и выпустила двухъядерную микросхему **Core 2 duo**. Когда возникла необходимость в более дешевой одноядерной версии микросхемы, фирма Intel стала продавать Core 2 duo с одним отключенным ядром, потому что небольшие дополнительные затраты на каждой производимой микросхеме обходились несравненно дешевле огромных затрат на проектирование и тестирование новой модели «с нуля». Серия Core продолжала развиваться; модели i3, i5 и i7 стали популярными решениями для низко-, средне- и высокопроизводительных компьютеров.

Несомненно, в будущем появятся и другие варианты. Фотография микросхемы i7 приводится на рис. 1.8. На ней в действительности размещено восемь ядер, но во всех версиях, кроме Xeon, включены только шесть. Такой подход означает, что микросхему с одним или двумя дефектными ядрами все равно можно продать — достаточно отключить дефектное ядро(-а). Каждое ядро имеет собственные кэши 1 и 2 уровня, но также имеется общий кэш 3 уровня (L3), используемый всеми ядрами. Кэши будут подробно рассмотрены позднее.

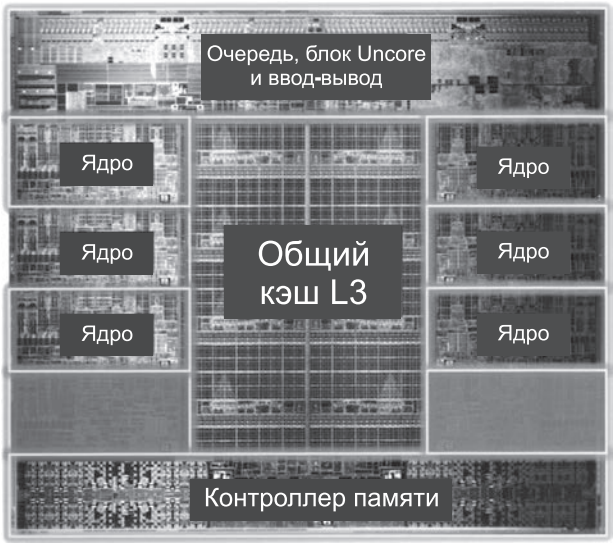


Рис. 1.8. Микросхема Intel Core i7-3960X. Подложка имеет размеры 21×21 мм и содержит 2,27 миллиарда транзисторов (фотография используется с разрешения корпорации Intel)

Помимо основной линейки процессоров, которую мы рассмотрели, Intel разрабатывает специальные варианты микросхем для отдельных сегментов рынка. В начале 1998 года компания запустила новую линейку под названием **Celeron**. По сути, это был дешевый вариант Pentium 2 с пониженной производительностью для низкопроизводительных компьютеров. Поскольку у процессора Celeron такая же архитектура, как у Pentium 2, мы не будем обсуждать его в этой книге. В июне 1998 года компания Intel выпустила специальную версию Pentium 2 для верхнего сегмента рынка — Xeon. Эту модификацию снабдили кэш-памятью большего объема, ускоренной внутренней шиной и усовершенствованными средствами поддержки мультипроцессорного режима, однако по всем остальным параметрам она ничем не отличалась от Pentium 2, что дает нам полное право не рассматривать ее отдельно. Для процессоров Pentium III, как и более поздних микросхем, также была разработана версия Xeon. На более новых процессорах одной из особенностей Xeon является увеличенное количество ядер.

В 2003 году появилась микросхема Pentium M (где M — сокращение от «Mobile») для портативных компьютеров. Она задумывалась как составная часть новой архитектуры Centrino, которая должна была, во-первых, снизить энергопотребление, увеличив тем самым ресурс аккумулятора, во-вторых, обеспечить

возможность производства более компактных и легких корпусов, в-третьих, предоставить встроенную поддержку беспроводных сетевых соединений по стандарту IEEE 802.11 (WiFi). Pentium M потребляет меньше энергии и отличается большей компактностью по сравнению с Pentium 4; вероятно, эти две характеристики вскоре позволят ему (и его преемникам) вытеснить микроархитектуру Pentium 4 в будущих продуктах Intel.

Все микросхемы Intel обратно совместимы со всеми своими предшественниками вплоть до модели 8086. Другими словами, программы, написанные когда-то для 8086, исполняются на Pentium 4 без каких бы то ни было изменений. Обратная совместимость в течение длительного времени является одним из основных принципов проектирования в Intel — соблюдение этого принципа позволяет сохранить предыдущие инвестиции в программное обеспечение. Естественно, поскольку модель Pentium 4 на три порядка сложнее, чем 8086, ее возможности несопоставимо шире. Из-за постепенных расширений, которые проектировщикам процессоров приходилось внедрять для достижения этой цели, архитектура получилась не такой элегантной, как если бы разработчики Pentium 4 получили 42 миллиона транзисторов и начали бы все строить «с нуля».

Интересно, что хотя закон Мура раньше ассоциировался с числом битов в памяти компьютера, он в равной степени применим и к процессорам. Если напротив даты выпуска каждой микросхемы поставить количество транзисторов на этой микросхеме по полулогарифмической шкале (см. табл. 1.4), мы увидим, что закон Мура действует и здесь. График показан на рис. 1.9.

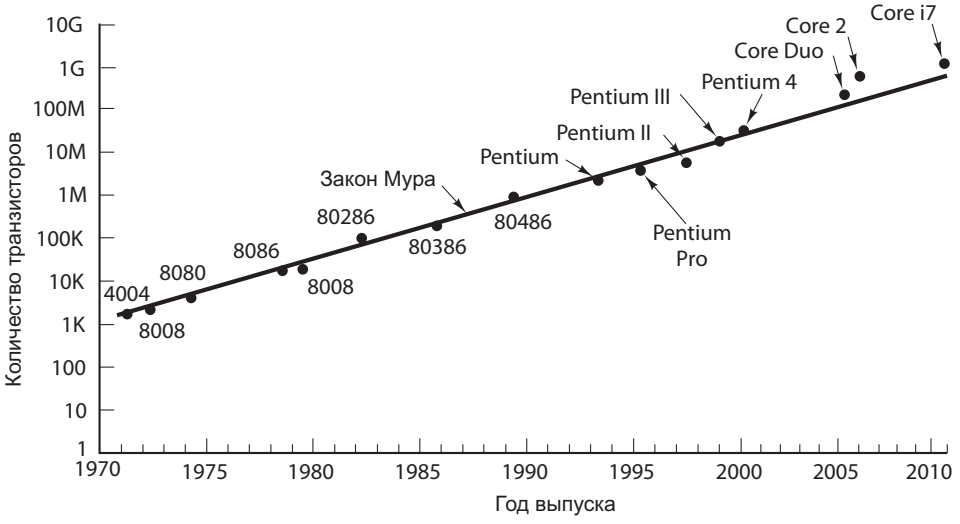


Рис. 1.9. Закон Мура действителен и для процессоров

Вероятно, закон Мура будет действовать еще несколько лет, однако уже сейчас начинает проявляться проблема, способная нарушить тенденцию — теплоотдача. В настоящее время тактовая частота повышается за счет уменьшения размера транзисторов, что, в свою очередь, вызывает потребность в более высоком напряжении. Потребление энергии и теплоотдача прямо пропорциональны

квадрату напряжения, а, значит, чем выше скорость, тем больше выделяется тепла, которое необходимо отводить. Процессор Pentium 4 с тактовой частотой 3,6 ГГц потребляет 115 Вт. При этом он выделяет примерно столько же тепла, сколько лампочка на 100 Вт. Чем больше повышается тактовая частота, тем заметнее становится проблема.

В ноябре 2004 компания Intel была вынуждена отменить выпуск модели Pentium 4 с тактовой частотой 4 ГГц из-за проблем с теплоотводом. Большие вентиляторы способны решить проблему, но они слишком шумные, что, естественно, не нравится пользователям. Водяное охлаждение, применяемое на мэйнфреймах, совершенно неприемлемо для настольных машин, не говоря уже о ноутбуках. Поэтому некогда безудержный рост тактовой частоты, вероятно, на некоторое время поуспокоится — по крайней мере, до того момента, когда инженеры Intel придумают эффективный способ отвода тепла. В планах Intel теперь другие новации — компания планирует разместить на одной микросхеме два процессора и снабдить ее общим кэшем большого объема. Поскольку величина энергопотребления определяется напряжением и тактовой частотой, два процессора на одной схеме потребляют значительно меньше энергии, чем один, работающий на удвоенной скорости. Таким образом, действие закона Мура может в будущем перейти с повышения тактовых частот на увеличение количества ядер и объема встроенных кэшей. Использование мультипроцессоров усложняет задачу программиста, потому что в отличие от изощренных однопроцессорных микроархитектур прошлого, способных выжать больше производительности из существующих программ, мультипроцессоры заставляют программиста явно управлять параллельным выполнением с использованием программных потоков, семафоров, общей памяти и других технологий — хлопотных и подверженных ошибкам.

Введение в архитектуру ARM

В начале 1980-х годов английская компания Acorn Computer на волне успеха своего 8-разрядного персонального компьютера BBC Micro приступила к работе над второй машиной, надеясь составить конкуренцию недавно выпущенному IBM PC. Компьютер BBC Micro был построен на базе 8-разрядного процессора 6502, и Стив Фарбер (Steve Furber) со своими коллегами из Acorn почувствовал, что 6502 не сможет соперничать с 16-разрядным процессором IBM PC 8086. Рассмотрев возможные альтернативы на рынке, они решили, что выбор слишком ограничен.

Под впечатлением проекта Berkeley RISC, в котором небольшая группа спроектировала на удивление быстрый процессор (на основе которого в конечном итоге была построена архитектура SPARC), они решили построить для своего проекта собственный процессор. Проект был назван Acorn RISC Machine (сокращенно ARM — позднее, после отделения ARM от Acorn, расшифровка была заменена на «Advanced RISC Machine»). Работа была завершена в 1985 году. Процессор использовал 32-разрядные команды и данные, имел 26-разрядное адресное пространство и производился фирмой VLSI Technology.

Первая архитектура ARM (названная ARM2) появилась в персональном компьютере Acorn Archimedes. Это была очень быстрая и недорогая для своего времени машина, которая выполняла до 2 MIPS (миллионов команд в секунду)

и стоила на момент выпуска всего 899 фунтов. Машина стала весьма популярной в Великобритании, Ирландии, Австралии и Новой Зеландии, особенно в школах.

Видя успех Archimedes, фирма Apple обратилась к Acorn с предложением разработать процессор ARM для своего нового проекта Apple Newton. Чтобы работа над проектом шла более целенаправленно, группа ARM покинула Acorn и создала новую компанию, названную Advanced RISC Machines (ARM). Их новый процессор ARM 610 устанавливался в Apple Newton на момент его выпуска в 1993 году. В отличие от исходной архитектуры ARM, новый процессор включал 4-килобайтный кэш, существенно повышавший производительность. Хотя Apple Newton не пользовался большим успехом, процессор ARM 610 нашел другие успешные применения; в частности, он использовался в компьютере Acorn RISC PC.

В середине 1990-х годов фирма ARM совместно с Digital Equipment Corporation разработала высокоскоростную версию ARM с пониженным энергопотреблением для устройств с ограниченным энергоресурсом — таких, как PDA. Они разработали архитектуру StrongARM, которая с первого появления вызвала в отрасли ажиотаж из-за своей высокой скорости (233 МГц) и сверхнизкой мощности (1 ватт). Эффективность обеспечивалась простой, четкой структурой, включавшей два 16-килобайтных кэша для команд и данных. StrongARM и его преемники в DEC пользовались умеренным коммерческим успехом. Они устанавливались во многих PDA, телевизионных абонентских приставках, мультимедийных устройствах и маршрутах.

Вероятно, самой известной из архитектур ARM стал процессор ARM7, который был выпущен ARM в 1994 году и продолжает широко использоваться в наши дни. Он содержит раздельные кэши команд и данных, а также реализует 16-разрядный набор команд Thumb — сокращенную версию полного 32-разрядного набора команд ARM, которая позволяет программировать многие стандартные операции в меньших 16-разрядных командах, существенно сокращая объем необходимой памяти. Процессор хорошо подходил для широкого диапазона низко- и среднепроизводительных встроенных систем, среди которых были тостеры, системы управления двигателем и даже портативное игровое устройство Nintendo Gameboy Advance.

В отличие от многих компьютерных компаний, ARM не производит микропроцессоры. Вместо этого фирма создает архитектуры, средства разработчика и библиотеки и продает лицензии на них разработчикам систем и производителям микросхем. Например, в планшетном компьютере Samsung Galaxy Tab на базе Android использовался процессор ARM. Galaxy Tab содержит однокристальный процессор Tegra 2, включающий два процессора ARM Cortex-A9 и графический процессор Nvidia GeForce. Ядра Tegra 2 были спроектированы ARM, интегрированы в однокристальную архитектуру Nvidia и выпущены TSMC (Taiwan Semiconductor Manufacturing Company). Перед нами впечатляющий пример сотрудничества компаний из разных стран, в котором каждая компания внесла свой вклад в конечный результат.

На рис. 1.10 представлена фотография подложки однокристальной системы Nvidia Tegra 2. Она состоит из трех процессоров ARM: двух ядер ARM Cortex 1,2 ГГц и ядра ARM7. Cortex-A9 — двухпоточное ядро с неупорядоченным

исполнением команд, оснащенное 1-мегабайтным кэшем L2 и поддержкой многопроцессорной обработки с разделением памяти. (Здесь много малопонятного технического жаргона, но мы доберемся до этих терминов в следующих главах. А пока достаточно знать, что эти особенности делают процессор очень быстрым!) ARM7 — меньшее и более старое ARM-ядро, используемое для конфигурации системы и управления питанием. Графическое ядро представляет собой 333-мегагерцовый графический процессор (GPU) GeForce, оптимизированный для работы на низкой мощности. Также в Tegra 2 включен кодировщик/декодировщик видео, аудиопроцессор и интерфейс видеовывода HDMI.

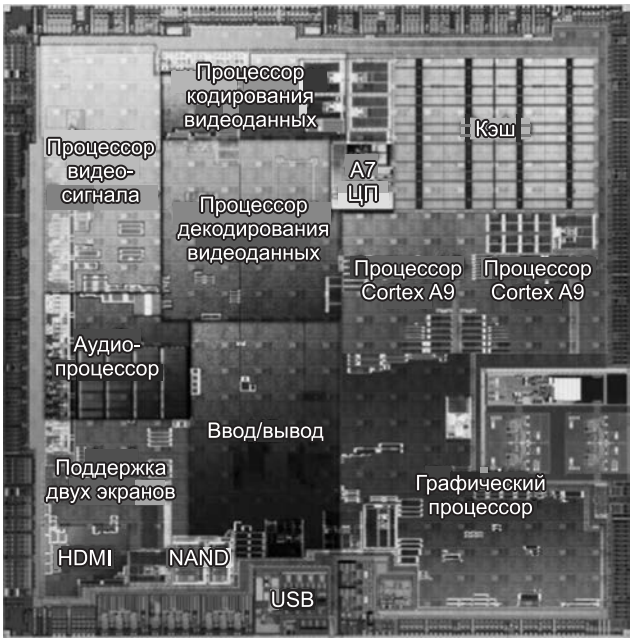


Рис. 1.10. Однокристальная система Nvidia Tegra 2
(фотография используется с разрешения корпорации Nvidia)

Архитектура ARM пользовалась огромным успехом в секторах пониженного энергопотребления, мобильных и встроенных систем. В январе 2011 года фирма ARM объявила о том, что продажи процессоров ARM с момента выпуска достигли 15 миллиардов и что продажи продолжают расти. Несмотря на то что архитектура ARM адаптирована для рынков низкопроизводительных устройств, она обладает вычислительным потенциалом для любого рынка, и некоторые признаки указывают на возможное расширение горизонта. Например, в октябре 2011 года был анонсирован 64-разрядный ARM-процессор. Также в январе 2011 года фирма Nvidia анонсировала «проект Denver» — однокристальная система на базе ARM для серверного и других рынков. Архитектура будет содержать несколько 64-разрядных ARM-процессоров с несколькими графическими процессорами общего назначения (GPGPU, General-Purpose GPU). Пониженное энергопотребление поможет снизить требования к средствам охлаждения серверных ферм и центров обработки данных.

Введение в архитектуру AVR

Наш третий пример очень сильно отличается от первого (архитектура x86, используемая в персональных компьютерах и серверах) и второго (архитектура ARM, используемая в PDA и смартфонах). Архитектура AVR используется в чрезвычайно низкопроизводительных встроенных системах. История AVR начинается в 1996 году в Норвежском технологическом институте, когда студенты Алф-Эгиль Боген (Alf-Egil Bogen) и Вегард Воллан (Vegard Wollan) спроектировали 8-разрядный RISC-процессор, названный AVR. По слухам, такое название было выбрано потому, что это был «RISC-процессор Алфа и Вегарда» ((A)lf and (V)egard's(R)ISC processor). Вскоре после завершения проектирования фирма Atmel купила разработку и открыла норвежское отделение, в котором два архитектора продолжали совершенствовать процессор AVR. Первый микроконтроллер AVR — AT90S1200 — был представлен Atmel в 1997 году. Чтобы упростить задачу проектировщиков систем, разводка контактов в точности соответствовала Intel 8051 — одного из самых популярных микроконтроллеров того времени. В наши дни к архитектуре AVR проявляется значительный интерес, потому что она заложена в основу чрезвычайно популярной платформы встроенных контроллеров Arduino.

Архитектура AVR реализована в трех классах микроконтроллеров, перечисленных в табл. 1.5. Модель низшего класса — tinyAVR — спроектирована для большинства приложений с жесткими ограничениями по размерам, мощности и затратам. Она содержит 8-разрядный процессор, простейшую поддержку цифрового ввода-вывода и поддержку аналогового ввода (например, чтение значений температуры с термистора). Модель tinyAVR настолько мала, что ее контакты имеют двойное назначение: они перепрограммируются во время выполнения для выполнения любых цифровых или аналоговых функций, поддерживаемых микроконтроллером. В модели megaAVR, используемой в популярной открытой встроенной системе Arduino, также добавлена поддержка последовательного ввода-вывода, внутренние часы и программируемый аналоговый вывод. Самой производительной моделью в этой низкопроизводительной категории является микроконтроллер AVR XMEGA, в котором также добавлен ускоритель криптографических операций и встроенная поддержка интерфейса USB.

Таблица 1.5. Классы микроконтроллеров семейства AVR

Микросхема	Флэш-память	EEPROM	Оперативная память	Контакты	Особенности
tinyAVR	0,5–16 Кбайт	0–512 байт	32–512 байт	6–32	Малые размеры, цифровой ввод-вывод, аналоговый ввод
megaAVR	8–256 Кбайт	0,5–4 Кбайт	0,25–8 Кбайт	28–100	Много периферийных устройств, аналоговый вывод
AVR XMEGA	16–256 Кбайт	1–4 Кбайт	2–16 Кбайт	44–100	Ускорение криптографический операций, ввод-вывод через USB

Кроме различных периферийных компонентов, каждый класс процессоров AVR включает дополнительные ресурсы памяти. Микроконтроллеры обычно оснащаются тремя видами памяти: флэш-памятью, перепрограммируемой постоянной памятью (EEPROM, Electrically Erasable Programmable Read Only Memory) и оперативной памятью (RAM, Random Access Memory). Флэш-память программируется через внешний интерфейс с использованием высоких напряжений; в этой памяти хранится код и данные программы. Флэш-память является энергонезависимой, так что даже в случае отключения системы ее содержимое сохраняется. Память EEPROM тоже является энергонезависимой, но в отличие от флэш-памяти она может изменяться программой во время выполнения. В этой памяти встроенная система хранит информацию о конфигурации — например, формат отображения времени в электронных часах (12- или 24-часовой). Наконец, в оперативной памяти хранятся переменные во время выполнения программы. Эта память не сохраняет информацию при выключении питания. Разные типы памяти подробно рассматриваются в главе 2.

Рецепт успеха в отрасли микроконтроллеров прост: разместите на микросхеме все, что только может понадобиться (хоть кухонную раковину, если ее удастся сократить до квадратного миллиметра), и оформите ее в виде недорогой и компактной микросхемы с небольшим количеством разъемов. Интеграция многих функций позволяет применять микроконтроллер для решения многих задач, а малые размеры и компактность обеспечивают его использование в разных форм-факторах. Чтобы вы лучше поняли, сколько разных функций включается в современный микроконтроллер, мы приведем список периферийных подсистем для Atmel megaAVR-168:

1. Три таймера (два 8-разрядных и один 16-разрядный).
2. Часы реального времени с тактовым генератором.
3. Шесть каналов широтно-импульсной модуляции, используемых, например, для управления интенсивностью света или скоростью двигателя.
4. Восемь каналов аналогово-цифровых преобразований, используемых для чтения уровней напряжения.
5. Универсальный последовательный приемник/передатчик.
6. Последовательный интерфейс I2C — единый стандарт взаимодействия с датчиками.
7. Программируемый сторожевой таймер, обнаруживающий блокировку системы.
8. Встроенный аналоговый компаратор, сравнивающий два входных напряжения.
9. Детектор снижения напряжения, прерывающий работу системы при сбоях питания.
10. Внутренний программируемый генератор для управления тактовой частотой процессора.

Единицы измерения

Во избежание недоразумений нелишне заметить, что в этой книге, равно как и в вычислительной технике в целом, вместо традиционных британских еди-

ниц используются метрические. Основные метрические приставки приведены в табл. 1.6. Префиксы обычно сокращаются по первым буквам, а названия единиц больше 1 записываются в верхнем регистре (Кбайт, Мбайт и т. д.). Таким образом, канал связи со скоростью 1 Мбит/с передает 10^6 бит в секунду, а 100-петасекундный генератор срабатывает каждые 10^{-10} секунд.

Таблица 1.6. Основные метрические приставки

Порядок	Явная величина	Приставка
10^{-3}	0,001	Милли
10^{-6}	0,000001	Микро
10^{-9}	0,000000001	Нано
10^{-12}	0,000000000001	Пико
10^{-15}	0,000000000000001	Фемто
10^{-18}	0,000000000000000001	Атто
10^{-21}	0,000000000000000000001	Зепто
10^{-24}	0,000000000000000000000001	Йокто
10^{3}	1000	Кило
10^{6}	1 000 000	Мега
10^{9}	1 000 000 000	Гига
10^{12}	1 000 000 000 000	Тера
10^{15}	1 000 000 000 000 000	Пета
10^{18}	1 000 000 000 000 000 000	Экса
10^{21}	1 000 000 000 000 000 000 000	Зета
10^{24}	1 000 000 000 000 000 000 000 000	Йотта

Следует иметь в виду, что при измерении емкости памяти, дисковых накопителей, файлов и баз данных в компьютерной отрасли вышеуказанные единицы измерения приобретают несколько другой смысл. Например, приставка *кило* означает не 10^3 (1000), а 2^{10} (1024). Иными словами, объем памяти всегда выражается степенью числа «2». Таким образом, в 1 Кбайт содержится 2^{10} (1024) байт, в 1 Мбайт — 2^{20} (1 048 576) байт, в 1 Гбайт — 2^{30} (1 073 741 824) байт, в 1 Тбайт — 2^{40} (1 099 511 627 776) байт.

С другой стороны, по каналу с пропускной способностью 1 Кбит/с за секунду передается 1000 бит, а в локальной сети на 10 Мбит/с — 10 000 000 бит. Это связано с тем, что пропускная способность не ограничена значениями, являющимися степенями двойки. К сожалению, многие люди путают эти две системы, особенно при оценке емкости дисковых накопителей.

Чтобы избежать двусмысленности, органы стандартизации ввели новые термины: *кибибайт* для 2^{10} байт, *мебибайт* для 2^{20} байт, *гибибайт* для 2^{30} и *тебибайт* для 2^{40} байт соответственно. Тем не менее особой популярности они пока не получили. Как нам кажется, до тех пор пока термины не станут общепринятыми, лучше использовать обозначения Кбайт, Мбайт, Гбайт и Тбайт для 2^{10}, 2^{20}, 2^{30}

и 2^{40} байт соответственно, а обозначения Кбит/с, Мбит/с, Гбит/с и Тбит/с — для 10^3, 10^6, 10^9 и 10^{12} бит/с.

Краткое содержание книги

Эта книга посвящена многоуровневым компьютерам и их организации (отметим, что почти все современные компьютеры многоуровневые). Подробно мы рассмотрим четыре уровня — цифровой логический уровень, уровень микроархитектуры, уровень архитектуры набора команд и уровень операционной системы. Основные вопросы, которые обсуждаются в этой книге, включают общую структуру уровней (и почему уровни построены именно таким образом), типы команд и данных, организацию памяти, адресацию, а также способы построения каждого уровня. Все это называется компьютерной организацией, или компьютерной архитектурой.

Мы в первую очередь имеем дело с общими понятиями и не касаемся деталей и строгой математики. По этой причине многие примеры значительно упрощены, чтобы сделать упор на основных понятиях, а не на деталях.

Чтобы разъяснить, как принципы, изложенные в этой книге, могут применяться на практике, мы в качестве примеров используем компьютеры x86, ARM и AVR. Они были выбраны по нескольким причинам. Во-первых, они широко используются, и у читателя наверняка есть доступ хотя бы к одному из них. Во-вторых, каждый из этих компьютеров обладает собственной уникальной архитектурой, что дает основу для сравнения и возможность показать альтернативные варианты. Книги, в которых рассматривается только один компьютер, оставляют у читателя чувство, будто это и есть единственный нормальный компьютер, что является абсурдным в свете огромного числа компромиссов и произвольных решений, которые разработчики вынуждены принимать. Читатель должен рассматривать эти и все другие компьютеры критически и стараться понять, почему дела обстоят именно таким образом и что можно изменить, а не просто принимать их как данность.

Нужно уяснить с самого начала, что эта книга не о том, как программировать x86, ARM и AVR. Эти компьютеры используются только в качестве иллюстративных примеров, и мы не претендуем на их полное описание. Читателям, желающим ознакомиться с этими компьютерами, следует обратиться к публикациям производителей.

Глава 2 знакомит читателей с основными компонентами компьютера: процессорами, памятью, устройствами ввода-вывода. В ней дается краткое описание системной архитектуры, что потребуется при чтении следующих глав.

Главы 3–6 касаются каждая одного из уровней, показанных на рис. 1.2. Мы идем снизу вверх, поскольку компьютеры разрабатывались именно таким образом. Структура уровня k в значительной степени определяется особенностями уровня k–1, поэтому очень трудно понять, как устроен определенный уровень, если не рассмотреть подробно предыдущий, который и определяет строение последующего. К тому же с точки зрения обучения логичнее следовать от более простых уровней к более сложным, а не наоборот.

Глава 3 посвящена цифровому логическому уровню, то есть аппаратному обеспечению. В ней рассказывается, что такое вентили и как они объединяются в схемы. В этой главе также вводятся основные понятия булевой алгебры, которая используется для обработки цифровых данных. Кроме того, объясняется, что такое шины, причем особое внимание уделяется популярной шине PCI. В главе приводится много разнообразных примеров, в том числе относящихся к трем упомянутым ранее компьютерам.

Глава 4 знакомит читателя со строением уровня микроархитектуры и принципами его работы. Поскольку функцией этого уровня является интерпретация команд второго уровня, мы сконцентрируемся именно на этом, что и проиллюстрируем на примерах. В этой главе также рассказывается о уровне микроархитектуры некоторых реальных систем.

В главе 5 обсуждается уровень архитектуры набора команд (ISA), который многие называют машинным языком. Здесь один из компьютеров, выбранных нами в качестве примеров, будет рассмотрен более подробно.

В главе 6 говорится о некоторых командах, об устройстве памяти компьютера, о механизмах управления на уровне операционной системы. В примерах фигурируют две операционные системы: Windows (популярная в настольных системах на базе x86) и UNIX, применяемая во многих системах на базе x86 и ARM.

Глава 7 — об уровне языка ассемблера. Сюда относится и язык ассемблер, и процесс ассемблирования. Здесь речь также идет о компоновке.

В главе 8 обсуждаются параллельные компьютеры, важность которых возрастает с каждым днем. Одни из них действуют на базе нескольких процессоров с общей памятью, у других общей памяти нет. Одни из них представляют собой суперкомпьютеры, другие — сети рабочих станций, третьи — системы на одной микросхеме.

Глава 9 содержит алфавитный список литературы, цитируемой в этой книге. Список рекомендуемой литературы размещен на веб-сайте книги по адресу www.prenhall.com/tanenbaum.

Вопросы и задания

1. Объясните следующие термины своими словами:
   1) транслятор;
   2) интерпретатор;
   3) виртуальная машина.
2. Может ли компилятор производить данные непосредственно для уровня микроархитектуры, минуя уровень архитектуры набора команд? Обсудите все доводы за и против.
3. Можете ли вы представить многоуровневый компьютер, у которого уровень физических устройств и цифровой логический уровень — не самые нижние уровни? Объясните, почему.
4. Рассмотрим многоуровневый компьютер, в котором все уровни отличаются друг от друга. Команды каждого уровня в m раз мощнее команд предыдущего

уровня, то есть одна команда уровня *r* может исполнять ту же работу, которую исполняют *m* команд на уровне r — 1. Если для исполнения программы первого уровня требуется *k* секунд, сколько времени будут исполняться соответствующие программы на уровнях 2, 3 и 4, учитывая, что для интерпретации одной команды уровня r + 1 требуется *n* команд уровня *r*?

5. Некоторые команды уровня операционной системы идентичны командам уровня архитектуры набора команд. Эти команды сразу исполняются микропрограммой, а не операционной системой. Учитывая ответ на предыдущий вопрос, подумайте, зачем это нужно.
6. Рассмотрим компьютер с идентичными интерпретаторами на уровнях 1, 2 и 3. Для выборки, анализа и исполнения одной команды интерпретатору требуется *n* служебных команд. На уровне 1 одна команда исполняется за *k* наносекунд. За какое время одна команда будет исполнена на уровнях 2, 3 и 4?
7. В каком смысле аппаратное и программное обеспечение эквивалентны? В каком они не эквивалентны?
8. Разностная машина Беббиджа была снабжена фиксированной программой без возможности внесения изменений. Не напоминает ли это вам современные компакт-диски? Аргументируйте свой ответ.
9. Одно из следствий идеи фон Неймана о хранении программы в памяти компьютера — возможность вносить изменения в программы. Приведите пример, где это может быть полезно (подсказка: подумайте об арифметических операциях над массивами).
10. Производительность 75-й модели 360 в 50 раз выше, чем модели 30, однако время цикла меньше всего лишь в 5 раз. Объясните, почему.
11. На рис. 1.4 и 1.5 изображены схемы компьютерных систем. Опишите, как происходит процесс ввода-вывода в каждой из систем. У какой из них общая производительность больше?
12. Предположим, что каждый из 300 миллионов жителей США каждый день потребляют две упаковки продуктов, оснащенных микросхемами RFID. Сколько таких микросхем придется произвести за год, чтобы покрыть такой объем спроса? Если одна микросхема стоит один цент, в какую сумму обойдется производство микросхем в полученном объеме? Сравните исчисленную сумму с объемом ВВП и подумайте, не станет ли проблематичным внедрение этой технологии?
13. Назовите три бытовых устройства, в которые имеет смысл устанавливать встроенные процессоры.
14. В определенный момент времени диаметр транзистора в микропроцессоре составлял один микрон. Каков будет диаметр транзистора в новой модели в следующем году в соответствии с законом Мура?
15. Ранее было показано, что закон Мура относится не только к плотности полупроводников, но и прогнозирует рост размеров компьютерных моделей (в разумных пределах) и сокращение времени моделирования. Покажите, что гидродинамическая модель, которая сегодня выполняется за 4 часа, будет выполняться за один час на компьютерах, построенных через 3 года, и всего

15 минут — на компьютерах, построенных через 6 лет. Затем покажите, что для крупномасштабной модели с ожидаемым временем выполнения в 5 лет расчеты завершатся раньше, если запустить их на 3 года позже.

16. В 1959 году компьютер IBM 7090 мог выполнять около 500 000 команд в секунду, его память состояла из 32 768 36-разрядных слов, а стоил он 3 миллиона долларов. Сравните его с современными компьютерами и вычислите сводный коэффициент их улучшения — для этого перемножьте отношения объемов памяти и скоростей, и разделите на отношение цен. Теперь посмотрите, к каким последствиям привели бы аналогичные улучшения в авиации за тот же период времени. «Боинг 707» начал выпускаться в существенных количествах в 1959 году. Его скорость составляла 950 км/час, изначально машина вмещала до 180 пассажиров и стоила 4 000 000 долларов. Какой скоростью, вместимостью и ценой обладал бы этот самолет сегодня, если бы авиация развивалась такими же темпами, что и компьютеры? Не забудьте привести свои оценки скорости, объема памяти и цены.
17. Развитие в компьютерной области часто имеет циклическую природу. Изначально наборы команд жестко программировались, затем они перешли на уровень микропрограмм, затем появились машины с RISC-процессорами, и они снова стали жестко программироваться. Изначально использовалась централизованная модель вычислений с большими компьютерами-мэйнфреймами. Укажите еще два направления развития, которые также демонстрируют цикличность.
18. Вопрос о том, кто является изобретателем компьютера, получил правовую оценку в апреле 1973 года, когда судья Эрл Ларсон вынес решение по иску Sperry Rand Corporation, владельца патентов на системы ENIAC, о нарушении патентных прав. Позиция компании Sperry Rand заключалась в том, что все без исключения производители компьютеров должны выплачивать ей роялти — по той простой причине, что ей принадлежали все основные патенты. Рассмотрение дела в суде началось в июне 1971 года; в общей сложности за период разбирательства суду было представлено свыше 30 000 документальных и вещественных доказательств. Стенограммы заседаний заняли более 20 000 страниц. Ваша задача состоит в том, чтобы как можно подробнее ознакомиться с материалами этого разбирательства, которые в изобилии представлены в Интернете, и написать отчет по технической составляющей дела. Что именно запатентовали Экерт и Моушли и почему судья решил, что их система была основана на более ранних разработках Атанасова?
19. Напишите краткое резюме о трех исследователях, которые, по вашему мнению, оказали наибольшее влияние на эволюцию аппаратного обеспечения компьютеров до их современного состояния; объясните, почему вы выбрали именно их.
20. Напишите аналогичное резюме относительно программного обеспечения.
21. Напишите краткое резюме о трех людях, которые, по вашему мнению, оказали наибольшее влияние на создание современных веб-сайтов, привлекающих большой объем трафика. Объясните, почему вы выбрали именно их.

Глава 2
Организация компьютерных систем

Цифровой компьютер состоит из связанных между собой процессоров, модулей памяти и устройств ввода-вывода. Глава 2 призвана познакомить читателя с этими компонентами и с тем, как они взаимосвязаны. Данная информация послужит основой для подробного рассмотрения каждого уровня в последующих пяти главах. Процессоры, память и устройства ввода-вывода — ключевые понятия, они будут упоминаться при обсуждении каждого уровня, поэтому изучение компьютерной архитектуры мы начнем с них.

Процессоры

На рис. 2.1 показана структура обычного компьютера с шинной организацией. **Центральный процессор** — это мозг компьютера. Его задача — выполнять программы, находящиеся в основной памяти. Для этого он вызывает команды из памяти, определяет их тип, а затем выполняет одну за другой. Компоненты соединены **шиной**, представляющей собой набор параллельно связанных проводов для передачи адресов, данных и управляющих сигналов. Шины могут быть внешними (связывающими процессор с памятью и устройствами ввода-вывода) и внутренними. Современный компьютер использует несколько шин.

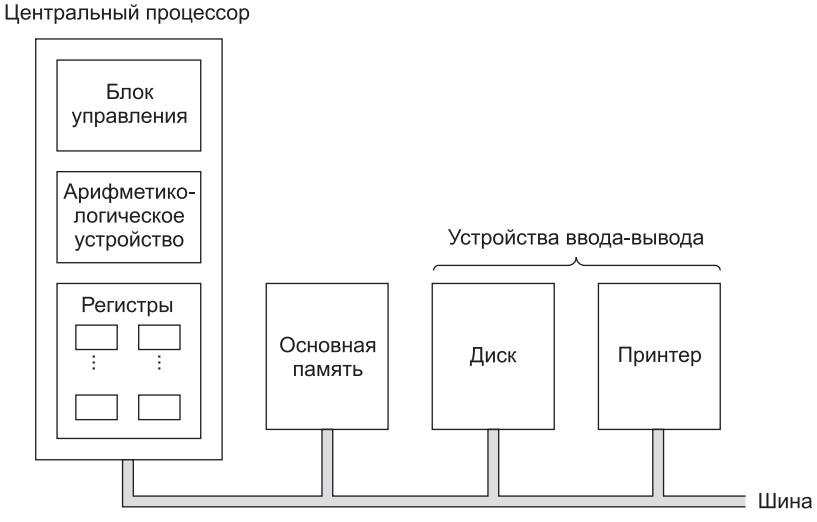


Рис. 2.1. Схема компьютера с одним центральным процессором и двумя устройствами ввода-вывода

Процессор состоит из нескольких частей. Блок управления отвечает за вызов команд из памяти и определение их типа. Арифметико-логическое устройство выполняет арифметические операции (например, сложение) и логические операции (например, логическое И).

Внутри центрального процессора находится быстрая память небольшого объема для хранения промежуточных результатов и некоторых команд управления. Эта память состоит из нескольких регистров, каждый из которых выполняет определенную функцию. Обычно размер всех регистров одинаков. Каждый регистр содержит одно число в диапазоне, верхняя граница которого зависит от размера регистра. Операции чтения и записи с регистрами выполняются очень быстро, поскольку они находятся внутри центрального процессора.

Самый важный регистр — **счетчик команд**, который указывает, какую команду нужно выполнять следующей. Название «счетчик команд» выбрано неудачно, поскольку он ничего не *считает*, но этот термин употребляется повсеместно[1]. Еще есть **регистр команд**, в котором находится выполняемая в данный момент команда. У большинства компьютеров имеются и другие регистры, одни из них многофункциональны, другие служат лишь какие-либо конкретным целям. Третьи регистры используются операционной системой для управления компьютером.

Устройство центрального процессора

Внутреннее устройство тракта данных типичного фон-неймановского процессора иллюстрирует рис. 2.2. **Тракт данных** состоит из регистров (обычно от 1 до 32), **арифметико-логического устройства (АЛУ)** и нескольких соединительных шин. Содержимое регистров поступает во входные регистры АЛУ, которые на рис. 2.2 обозначены буквами A и B. В них находятся входные данные АЛУ, пока АЛУ производит вычисления. Тракт данных — важная составная часть всех компьютеров, и мы обсудим его очень подробно.

АЛУ выполняет сложение, вычитание и другие простые операции над входными данными и помещает результат в выходной регистр. Содержимое этого выходного регистра может записываться обратно в один из регистров или сохраняться в памяти, если это необходимо. Не во всех архитектурах есть регистры A, B и выходные регистры. На рис. 2.2 представлена операция сложения, но АЛУ может выполнять и другие операции.

Большинство команд можно разделить на две группы: команды типа регистр-память и типа регистр-регистр. Команды первого типа вызывают слова из памяти, помещают их в регистры, где они используются в качестве входных данных АЛУ (слова — это такие элементы данных, которые перемещаются между памятью и регистрами[2]). Словом может быть целое число. Организацию памяти мы обсудим далее в этой главе. Другие команды этого типа помещают регистры обратно в память.

Команды второго типа вызывают два операнда из регистров, помещают их во входные регистры АЛУ, выполняют над ними какую-нибудь арифметическую или логическую операцию и переносят результат обратно в один из реги-

[1] Используется также термин «указатель команд». — *Примеч. науч. ред.*

[2] На самом деле размер слова обычно соответствует разрядности регистра данных. Так, у 16-разрядных микропроцессоров 8086 и 8088 слово имеет длину 16 бит, а у 32-разрядных микропроцессоров — 32 бита. — *Примеч. науч. ред.*

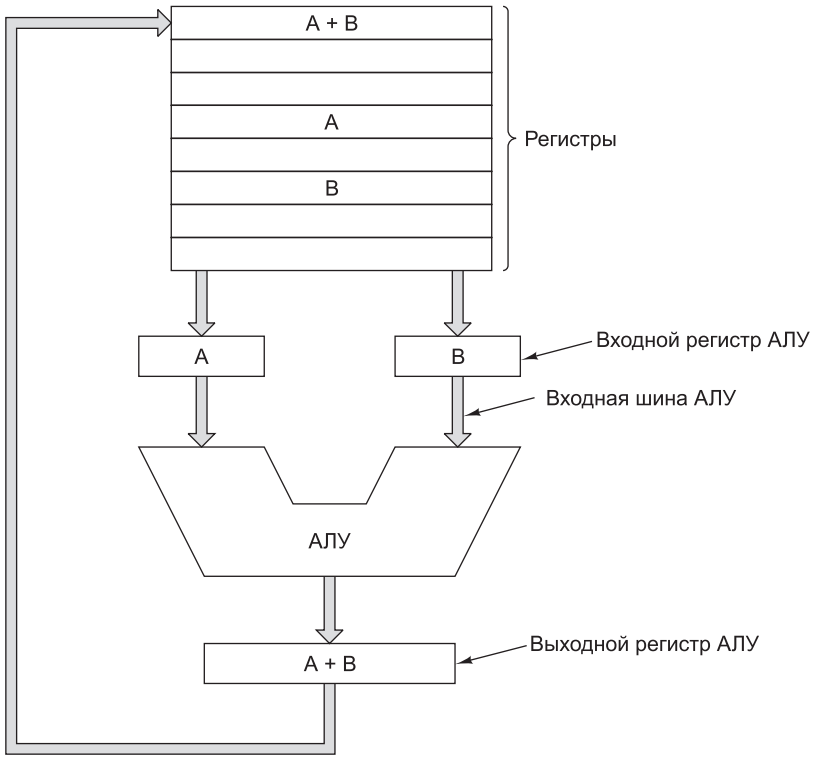


Рис. 2.2. Тракт данных обычной фон-неймановской машины

стров. Этот процесс называется **циклом тракта данных**. В какой-то степени он определяет, что может делать машина. Современные компьютеры оснащаются несколькими АЛУ, работающими параллельно и специализирующимися на разных функциях. Чем быстрее происходит цикл тракта данных, тем быстрее компьютер работает.

Выполнение команд

Центральный процессор выполняет каждую команду за несколько шагов. Он делает следующее:

1. Вызывает следующую команду из памяти и переносит ее в регистр команд.
2. Меняет положение счетчика команд, который после этого указывает на следующую команду[1].
3. Определяет тип вызванной команды.
4. Если команда использует слово из памяти, определяет, где находится это слово.
5. Переносит слово, если это необходимо, в регистр центрального процессора[2].

[1] Это происходит после декодирования текущей команды, а иногда и после ее выполнения. — *Примеч. науч. ред.*

[2] Следует заметить, что бывают команды, которые требуют загрузки из памяти целого множества слов и их обработки в рамках единственной команды. — *Примеч. науч. ред.*

6. Выполняет команду.
7. Переходит к шагу 1, чтобы начать выполнение следующей команды.

Такая последовательность шагов (**выборка** — **декодирование** — **исполнение**) является основой работы всех компьютеров.

Описание работы центрального процессора можно представить в виде программы. В листинге 2.1 приведена такая программа-интерпретатор на языке Java. В описываемом компьютере есть два регистра: счетчик команд (PC) с адресом следующей команды и аккумулятор (AC), в котором хранятся результаты арифметических операций. Кроме того, имеются внутренние регистры, в которых хранится текущая команда (`instr`), тип текущей команды (`instr_type`), адрес операнда команды (`data_loc`) и сам операнд (`data`). Каждая команда содержит один адрес ячейки памяти. В ячейке памяти хранится операнд — например, фрагмент данных, который нужно добавить в аккумулятор.

Листинг 2.1. Интерпретатор для простого компьютера (на языке Java)

```
public class Interp{
  static int PC;         // PC содержит адрес следующей команды
  static int AC;         // Аккумулятор, регистр для арифметики
  static int instr;      // Регистр для текущей команды
  static int instr_type; // Тип команды (код операции)
  static int data_loc;   // Адрес данных или -1, если его нет
  static int data;       // Текущий операнд
  static boolean run_bit = true; // Бит, который можно сбросить,
                                 // чтобы остановить машину
  public static void interpret(int memory[], int starting_address) {
// Эта процедура интепретирует программы для простой машины,
// которая содержит команды только с одним операндом из
// памяти. Машина имеет регистр AC (аккумулятор). Он
// используется для арифметических действий - например,
// команда ADD суммирует число из памяти с AC. Интерпретатор
// работает до тех пор, пока не будет выполнена команда
// HALT, вследствие чего бит run_bit поменяет значение на
// false. Машина состоит из блока памяти, счетчика команд, бита
// run bit и аккумулятора AC. Входные параметры представляют собой
// копию содержимого памяти и начальный адрес.

    PC=starting_address;
    while (run_bit) {
      instr=memory[PC]; // Вызывает следующую команду в instr
      PC=PC+1;          // Увеличивает значение счетчика команд
      instr_type=get_instr_type(instr);      // Определяет тип команды
      data_loc=find_data(instr, instr_type); //  Находит данные (-1,
                                             // если данных нет)
      if(data_loc>=0)   // Если data_loc=-1, значит, операнда нет
        data=memory[data_loc];               // Вызов данных
      execute(instr_type,data);              // Выполнение команды
    }
  }

  private static int get_instr_type(int addr) {...}
  private static int find_data(int instr, int type) {...}
  private static void execute(int type, int data) {...}
}
```

Сам факт того, что можно написать программу, имитирующую работу центрального процессора, показывает, что программа не обязательно должна выполняться реальным процессором (устройством). Напротив, вызывать из памяти, определять тип команд и выполнять эти команды может другая программа. Такая программа называется **интерпретатором**. Об интерпретаторах мы говорили в главе 1.

Эквивалентность аппаратных процессоров и интерпретаторов имеет важные последствия для организации компьютера и проектирования компьютерных систем. После того как разработчик выбрал машинный язык (Я) для нового компьютера, он должен решить, разрабатывать ли ему процессор, который будет выполнять программы на языке Я, или написать специальную программу для интерпретации программ на том же языке. Если он решит написать интерпретатор, ему потребуется разработать аппаратное обеспечение для исполнения этого интерпретатора. Возможны также гибридные конструкции, когда часть команд выполняется аппаратным обеспечением, а часть интерпретируется.

Интерпретатор разбивает команды на более мелкие (элементарные). В результате машина, предназначенная для исполнения интерпретатора, может быть гораздо проще по строению и дешевле, чем процессор, выполняющий программы без интерпретации. Такая экономия особенно важна при большом количестве сложных команд с различными параметрами. В сущности, экономия проистекает из самого факта замены аппаратного обеспечения программой (интерпретатором), тогда как создание копий программного продукта обходится дешевле, чем создание копий аппаратных элементов.

Первые компьютеры поддерживали небольшое количество команд, и эти команды были простыми. Однако разработка более мощных компьютеров привела, помимо всего прочего, к появлению более сложных команд. Вскоре разработчики поняли, что при наличии сложных команд программы выполняются быстрее, хотя выполнение каждой отдельной команды занимает больше времени. (В качестве примеров таких сложных команд можно назвать выполнение операций с плавающей точкой, обеспечение прямого доступа к элементам массива и т. п.) Если обнаруживалось, что пара тех или иных команд часто выполняется последовательно, нередко вводилась новая команда, заменяющая эти две.

Сложные команды оказались лучше еще и потому, что некоторые операции иногда перекрывались. Подобные операции могли выполняться параллельно, но для этого требовалась дополнительная аппаратура. Для дорогих компьютеров с высокой производительностью приобретение такого дополнительного аппаратного обеспечения было вполне оправданным. Таким образом, у дорогих компьютеров было гораздо больше команд, чем у дешевых. Однако растущая стоимость разработки и требования совместимости команд привели к тому, что сложные команды стали использоваться и в дешевых компьютерах, хотя там во главу угла ставилась стоимость, а не быстродействие.

К концу 50-х годов компания IBM, которая лидировала тогда на компьютерном рынке, решила, что производство семейства компьютеров, каждый из которых выполняет одни и те же команды, выгоднее и для самой компании, и для покупателей. Чтобы охарактеризовать этот уровень совместимости, компания IBM ввела термин **архитектура**. Новое семейство компьютеров должно было иметь единую архитектуру и много разных моделей, отличающихся по цене и скорости,

но «умеющих» выполнять одни и те же программы. Но как построить дешевый компьютер, который сможет выполнять все сложные команды, предназначенные для высокоэффективных дорогостоящих машин?

Решением проблемы стала интерпретация. Эта технология, впервые предложенная Уилксом в 1951 году, позволяла разрабатывать простые дешевые компьютеры, которые, тем не менее, могли выполнять большое количество команд. В результате компания IBM создала архитектуру System/360, семейство совместимых компьютеров, различающихся по цене и производительности почти на два порядка. Аппаратное обеспечение, позволяющее работать без интерпретации, использовалось только в самых дорогих моделях.

Простые компьютеры с интерпретаторами команд имели свои достоинства. Наиболее важными среди них являлись:

- возможность исправлять неправильно реализованные команды «на месте» или даже компенсировать ошибки аппаратного обеспечения на уровне программного обеспечения;
- возможность добавлять новые команды при минимальных затратах, причем при необходимости уже после покупки компьютера;
- возможность (благодаря структурированной организации) разработки, проверки и документирования сложных команд.

В 70-е годы компьютерный рынок быстро разрастался, новые компьютеры могли выполнять все больше и больше функций. Вследствие повышенного спроса на дешевые компьютеры предпочтение отдавалось компьютерам с интерпретаторами. Возможность разрабатывать аппаратное обеспечение с интерпретатором для определенного набора команд привела к появлению дешевых процессоров. Полупроводниковые технологии быстро развивались, низкая стоимость брала верх над высокой производительностью, и интерпретаторы стали применяться при разработке компьютеров все шире и шире. Интерпретация использовалась практически во всех компьютерах, выпущенных в 70-е годы, от мини-компьютеров до самых больших машин.

К концу 70-х годов интерпретаторы стали применяться практически во всех моделях, кроме самых дорогих машин с очень высокой производительностью (например, Cray-1 и компьютеров серии Control Data Cyber). Интерпретаторы обеспечивали реализацию сложных команд без использования дорогостоящей аппаратуры, поэтому разработчики могли вводить все более и более сложные команды, а также (и даже в особенности) расширять способы определения операндов.

Эта тенденция достигла своего апогея в компьютере VAX, разработанном компанией DEC; у него было несколько сот команд и более 200 способов определения операндов в каждой команде. К несчастью, архитектура VAX с самого начала ориентировалась на интерпретацию, а производительности уделялось мало внимания. Это привело к появлению большого количества второстепенных команд, которые сложно было выполнять непосредственно. Данное упущение стало фатальным как для VAX, так и для его производителя (компании DEC). Компания Compaq купила DEC в 1998 году (правда, тремя годами позже сама компания Compaq вошла в структуру Hewlett-Packard).

Хотя самые первые 8-разрядные микропроцессоры были очень простыми и поддерживали небольшой набор команд, к концу 70-х годов даже они стали

разрабатываться с ориентацией на интерпретаторы. В этот период основной проблемой для разработчиков стала возрастающая сложность микропроцессоров. Главное преимущество интерпретации заключалось в том, что можно было разработать очень простой процессор, а вся самое сложное реализовать с помощью интерпретатора. Таким образом, вместо разработки сложной аппаратуры требовалась разработка сложного программного обеспечения.

Успех системы Motorola 68000 с большим набором интерпретируемых команд и одновременный провал компьютера Zilog Z8000, у которого был столь же обширный набор команд, но не было интерпретатора, продемонстрировали все преимущества интерпретации при разработке новых машин. Этот успех был довольно неожиданным, учитывая, что компьютер Z80 (предшественник Zilog Z8000) пользовался большей популярностью, чем Motorola 6800 (предшественник Motorola 68000). Конечно, важную роль здесь сыграли и другие факторы — например то, что компания Motorola много лет занималась производством микросхем, а торговая марка Zilog принадлежала Exxon — крупной нефтяной компании.

Еще один фактор в пользу интерпретации — существование быстродействующих постоянных запоминающих устройств для хранения интерпретаторов (так называемых командных ПЗУ). Предположим, что для выполнения обычной интерпретируемой команды интерпретатору компьютера Motorola 68000 нужно выполнить 10 команд (они называются **микрокомандами**), по 100 нс на каждую, и произвести два обращения к оперативной памяти, по 500 нс на каждое. Общее время выполнения команды составит, следовательно, 2000 нс — всего лишь в два раза больше, чем в лучшем случае заняло бы непосредственное выполнение этой команды без интерпретации. А если бы не было специального быстродействующего постоянного запоминающего устройства, выполнение этой команды заняло бы целых 6000 нс. Шестикратное возрастание времени выполнения вынести намного сложнее.

Системы RISC и CISC

В конце 70-х годов проводилось много экспериментов с очень сложными командами, появление которых стало возможным благодаря интерпретации. Разработчики пытались уменьшить «семантический разрыв» между тем, что компьютеры способны делать, и тем, что требуют языки высокого уровня. Едва ли кто-нибудь тогда думал о разработке более простых машин, так же как сейчас мало кто (к сожалению) занимается разработкой менее мощных электронных таблиц, сетей, веб-серверов и т. д.

В компании IBM этой тенденции противостояла группа разработчиков во главе с Джоном Коком (John Cocke); они попытались воплотить идеи Сеймура Крея, создав экспериментальный высокоэффективный мини-компьютер **801**. Хотя компания IBM не занималась сбытом этой машины, а результаты эксперимента были опубликованы только через несколько лет [Radin, 1982], весть быстро разнеслась по свету, и другие производители тоже занялись разработкой подобных архитектур.

В 1980 году группа разработчиков в университете Беркли во главе с Дэвидом Паттерсоном (David Patterson) и Карло Секвином (Carlo Séquin) начала раз-

работку не ориентированных на интерпретацию процессоров VLSI [Patterson, 1985; Patterson and Séquin, 1982]. Для обозначения этого понятия они придумали термин **RISC**, а новый процессор назвали RISC I, вслед за которым вскоре был выпущен RISC II. Немного позже, в 1981 году, Джон Хеннеси (John Hennesy) в Стенфорде разработал и выпустил другую микросхему, которую он назвал **MIPS** [Hennessy, 1984]. Эти две микросхемы развились в коммерчески важные продукты SPARC и MIPS соответственно.

Новые процессоры существенно отличались от коммерческих процессоров того времени. Поскольку они были несовместимы с существующей продукцией, разработчики вправе были включать туда новые наборы команд, которые могли бы повысить общую производительность системы. Первоначально основное внимание уделялось простым командам, которые могли быстро выполняться. Однако вскоре разработчики осознали, что ключом к высокой производительности компьютера является разработка команд, которые можно быстро *запускать*. То есть не так важно, как долго выполняется та или иная команда, важнее то, сколько команд в секунду может быть запущено.

В то время когда разрабатывались эти простые процессоры, всеобщее внимание привлекало относительно небольшое количество команд (обычно около 50). Для сравнения: число команд в компьютерах VAX производства DEC и больших компьютерах производства IBM в то время составляло от 200 до 300. Компьютер RISC (Reduced Instruction Set Computer — **компьютер с сокращенным набором команд**) противопоставлялся системе CISC (Complex Instruction Set Computer — **компьютер с полным набором команд**) — слабо завуалированный намек на компьютер VAX, который доминировал в то время в университетской среде. На сегодняшний день мало кто считает, что размер набора команд так уж важен, но названия сохранились до сих пор.

С этого момента началась грандиозная идеологическая война между сторонниками RISC и «консерваторами» (VAX, Intel, мэйнфреймы IBM). По мнению первых, наилучший способ разработки компьютеров — включение туда небольшого количества простых команд, каждая из которых выполняется за один цикл тракта данных (см. рис. 2.2), то есть производит над парой регистров какую-либо арифметическую или логическую операцию (например, сложение или операцию логического И) и помещает результат обратно в регистр. В качестве аргумента они утверждали, что даже если системе RISC приходится выполнять 4 или 5 команд вместо одной, которую выполняет CISC, RISC все равно выигрывает в скорости, так как RISC-команды выполняются в 10 раз быстрее (поскольку они не интерпретируются). Следует также отметить, что к этому времени быстродействие основной памяти приблизилась к быстродействию специальных командных ПЗУ, потому недостатки интерпретации были налицо, что еще более поднимало популярность компьютеров RISC.

Учитывая преимущества RISC в плане производительности, можно было предположить, что на рынке такие компьютеры, как UltraSPARC компании Sun, должны доминировать над компьютерами CISC (Intel Pentium и т. д.). Однако ничего подобного не произошло. Почему?

Во-первых, компьютеры RISC несовместимы с другими моделями, а многие компании вложили миллиарды долларов в программное обеспечение для продукции Intel. Во-вторых, как ни странно, компания Intel сумела воплотить те же

идеи в архитектуре CISC. Процессоры Intel, начиная с процессора 486, содержат RISC-ядро, которое выполняет самые простые (и обычно самые распространенные) команды за один цикл тракта данных, а по обычной технологии CISC интерпретируются более сложные команды. В результате обычные команды выполняются быстро, а более сложные и редкие — медленно. Хотя при таком «гибридном» подходе производительность ниже, чем в архитектуре RISC, новая архитектура CISC имеет ряд преимуществ, поскольку позволяет использовать старое программное обеспечение без изменений.

Принципы проектирования современных компьютеров

Прошло уже более двадцати лет с тех пор, как были сконструированы первые компьютеры RISC, однако некоторые принципы их функционирования можно перенять, учитывая современное состояние технологии разработки аппаратного обеспечения. Если происходит очень резкое изменение в технологии (например, новый процесс производства делает время обращения к памяти в 10 раз меньше, чем время обращения к центральному процессору), меняются все условия. Поэтому разработчики всегда должны учитывать возможные технологические изменения, которые могли бы повлиять на баланс между компонентами компьютера.

Существует ряд принципов разработки, иногда называемых **принципами RISC**, которым по возможности стараются следовать производители универсальных процессоров. Из-за некоторых внешних ограничений, например требования совместимости с другими машинами, приходится время от времени идти на компромисс, но эти принципы — цель, к которой стремится большинство разработчиков.

- *Все команды должны выполняться непосредственно аппаратным обеспечением*. То есть обычные команды выполняются напрямую, без интерпретации микрокомандами. Устранение уровня интерпретации повышает скорость выполнения большинства команд. В компьютерах типа CISC более сложные команды могут разбиваться на несколько шагов, которые затем выполняются как последовательность микрокоманд. Эта дополнительная операция снижает быстродействие машины, но может использоваться для редко применяемых команд.
- *Компьютер должен запускать как можно больше команд в секунду*. В современных компьютерах используется много различных способов повышения производительности, главный из которых — запуск как можно большего количества команд в секунду. В конце концов, если процессор сможет запустить 500 млн команд в секунду, то его производительность составляет 500 MIPS, сколько бы времени ни занимало выполнение этих команд. (**MIPS** — сокращение от Millions of Instructions Per Second — миллионов команд в секунду.) Этот принцип предполагает, что параллелизм должен стать важным фактором повышения производительности, поскольку запустить на выполнение большое количество команд за короткий промежуток времени можно только в том случае, если есть возможность одновременного выполнения нескольких команд.

  Хотя команды любой программы всегда располагаются в памяти в определенном порядке, компьютер изменить порядок их запуска (так как необходимые ресурсы памяти могут быть заняты) и (или) завершения. Конечно,

если команда 1 устанавливает значение в регистр, а команда 2 использует этот регистр, нужно действовать с особой осторожностью, чтобы команда 2 не считала значение из регистра раньше, чем оно там окажется. Чтобы не допускать подобных ошибок, необходимо хранить в памяти большое количество дополнительной информации, но благодаря возможности выполнять несколько команд одновременно производительность все равно оказывается выше.

- *Команды должны легко декодироваться*. Предел количества запускаемых в секунду команд зависит от темпа декодирования отдельных команд. Декодирование команд позволяет определить, какие ресурсы им необходимы и какие действия нужно выполнить. Все, что способствует упрощению этого процесса, полезно. Например, можно использовать единообразные команды с фиксированной длиной и с небольшим количеством полей. Чем меньше разных форматов команд, тем лучше.
- *К памяти должны обращаться только команды загрузки и сохранения*. Один из самых простых способов разбить операцию на отдельные шаги — сделать так, чтобы операнды большей части команд брались из регистров и возвращались туда же. Операция перемещения операндов из памяти в регистры и обратно может осуществляться в разных командах. Поскольку доступ к памяти занимает много времени, длительность которой невозможно спрогнозировать, выполнение этих команд могут взять на себя другие команды, единственное назначение которых — перемещение операндов между регистрами и памятью. То есть к памяти должны обращаться только команды загрузки и сохранения (`LOAD` и `STORE`).
- *Регистров должно быть много*. Поскольку доступ к памяти происходит относительно медленно, в компьютере должно быть много регистров (по крайней мере 32). Если слово было однажды загружено из памяти, при наличии большого числа регистров оно может содержаться в регистре до тех пор, пока не потребуется. Возвращение слова из регистра в память и новая загрузка этого же слова в регистр нежелательны. Лучший способ избежать излишних перемещений — наличие достаточного количества регистров.

Параллелизм на уровне команд

Разработчики компьютеров стремятся к тому, чтобы повысить производительность своих машин. Один из способов заставить процессоры работать быстрее — повышение их тактовой частоты, однако при этом существуют некоторые технологические ограничения на то, что можно сделать методом «грубой силы» на данный момент. Поэтому большинство проектировщиков для повышения производительности при данной тактовой частоте процессора используют параллелизм (выполнение двух или более операций одновременно).

Существует две основные формы параллелизма: параллелизм на уровне команд и параллелизм на уровне процессоров. В первом случае параллелизм реализуется за счет запуска большого количества команд каждую секунду. Во втором случае над одним заданием работают одновременно несколько процессоров. Каждый подход имеет свои преимущества. В этом разделе мы рассмотрим параллелизм на уровне команд, а в следующем — параллелизм на уровне процессоров.

Конвейеры

Уже много лет известно, что главным препятствием высокой скорости выполнения команд является необходимость их загрузки из памяти. Для разрешения этой проблемы можно вызывать команды из памяти заранее и хранить в специальном наборе регистров. Эта идея использовалась еще в 1959 году при разработке компьютера Stretch компании IBM, а набор регистров был назван **буфером выборки с упреждением**. Таким образом, когда требовалась определенная команда, она вызывалась прямо из буфера, а обращения к памяти не происходило.

В действительности при выборке с упреждением команда обрабатывается за два шага: сначала происходит выборка команды, а затем ее выполнение. Дальнейшим развитием этой стратегии стала концепция **конвейера**. При использовании конвейера команда обрабатывается уже не за два, а за большее количество шагов, каждый из которых реализуется определенным аппаратным компонентом, причем все эти компоненты могут работать параллельно.

На рис. 2.3, *а* изображен конвейер из 5 блоков, которые называются **ступенями**. Первая ступень (блок C1) вызывает команду из памяти и помещает ее в буфер, где она хранится до тех пор, пока не потребуется. Вторая ступень (блок C2) декодирует эту команду, определяя ее тип и тип ее операндов. Третья ступень (блок C3) определяет местонахождение операндов и вызывает их из регистров или из памяти. Четвертая ступень (блок C4) выполняет команду, обычно проводя операнды через тракт данных (см. рис. 2.2). И наконец, блок C5 записывает результат обратно в нужный регистр.

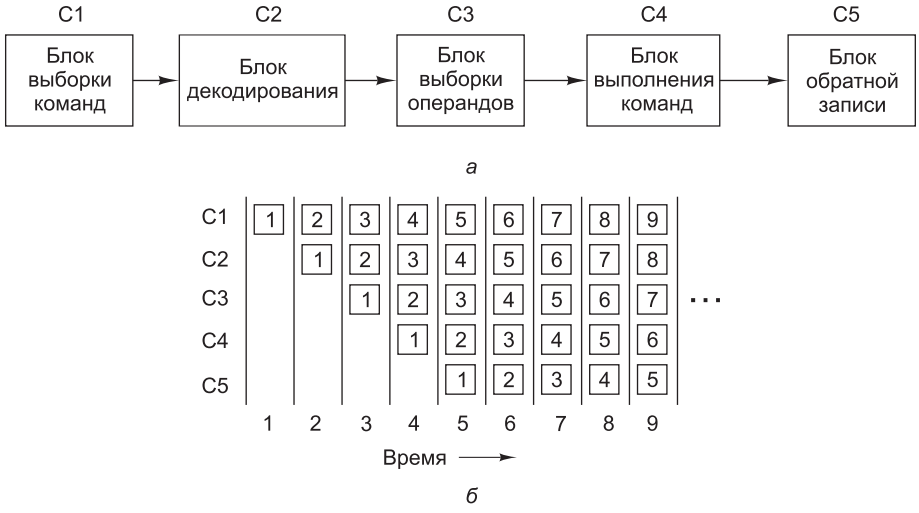


Рис. 2.3. Пятиступенчатый конвейер (*а*); состояние каждой ступени в зависимости от количества пройденных циклов (*б*). Показано 9 циклов

На рис. 2.3, *б* мы видим, как действует конвейер во времени. Во время цикла 1 блок C1 обрабатывает команду 1, вызывая ее из памяти. Во время цикла 2 блок C2 декодирует команду 1, в то время как блок C1 вызывает из памяти команду 2. Во время цикла 3 блок C3 вызывает операнды для команды 1, блок C2 декодирует команду 2, а блок C1 вызывает команду 3. Во время цикла 4 блок C4

выполняет команду 1, С3 вызывает операнды для команды 2, С2 декодирует команду 3, а С1 вызывает команду 4. Наконец, во время цикла 5 блок С5 записывает результат выполнения команды 1 обратно в регистр, тогда как другие ступени конвейера обрабатывают следующие команды.

Чтобы лучше понять принципы работы конвейера, рассмотрим аналогичный пример. Представим себе кондитерскую фабрику, на которой выпечка тортов и их упаковка для отправки производятся раздельно. Предположим, что в отделе отправки находится длинный конвейер, вдоль которого стоят 5 рабочих (или ступеней обработки). Каждые 10 секунд (это время цикла) первый рабочий ставит пустую коробку для торта на ленту конвейера. Эта коробка отправляется ко второму рабочему, который кладет в нее торт. После этого коробка с тортом доставляется третьему рабочему, который закрывает и запечатывает ее. Затем она поступает к четвертому рабочему, который ставит на ней штамп. Наконец, пятый рабочий снимает коробку с конвейерной ленты и помещает ее в большой контейнер для отправки в супермаркет. Примерно таким же образом действует компьютерный конвейер: каждая команда (в случае с кондитерской фабрикой — торт) перед окончательным выполнением проходит несколько ступеней обработки.

Возвратимся к нашему конвейеру на рис. 2.3. Предположим, что время цикла у этой машины — 2 нс. Тогда для того, чтобы одна команда прошла через весь конвейер, требуется 10 нс. На первый взгляд может показаться, что такой компьютер будет выполнять 100 млн команд в секунду, в действительности же скорость его работы гораздо выше. В течение каждого цикла (2 нс) завершается выполнение одной новой команды, поэтому машина выполняет не 100, а 500 млн команд в секунду!

Конвейеры позволяют добиться компромисса между **временем запаздывания** (время выполнения одной команды) и **пропускной способностью процессора** (количество команд, выполняемых процессором в секунду). Если время обращения составляет T нс, а конвейер имеет n ступеней, время запаздывания составит nT н.

Поскольку одна команда выполняется за одно обращение, а за одну секунду таких обращений набирается $10^9/T$, количество команд в секунду также составляет $10^9/T$. Скажем, если T = 2 нс, то каждую секунду выполняется 500 млн команд. Для того чтобы получить значение MIPS, нужно разделить скорость исполнения команд на один миллион; таким образом, $(10^9/T)/10^6 = 1000/T$ MIPS. В принципе, скорость исполнения команд можно измерять и в миллиардах операций в секунду (Billion Instructions Per Second, BIPS), но так никто не делает, и мы не будем.

Суперскалярные архитектуры

Один конвейер — хорошо, а два — еще лучше. Одна из возможных схем процессора с двумя конвейерами показана на рис. 2.4. В ее основе лежит конвейер, изображенный на рис. 2.3. Здесь общий блок выборки команд вызывает из памяти сразу по две команды и помещает каждую из них в один из конвейеров. Каждый конвейер содержит АЛУ для параллельных операций. Чтобы выполняться параллельно, две команды не должны конфликтовать из-за ресурсов (например, регистров) и ни одна из них не должна зависеть от результата выполнения другой. Как и в случае с одним конвейером, либо компилятор должен гарантировать

отсутствие нештатных ситуаций (когда, например, аппаратура не обеспечивает проверку команд на несовместимость и при обработке таких команд выдает некорректный результат), либо конфликты должны выявляться и устраняться дополнительным оборудованием непосредственно в ходе выполнения команд.

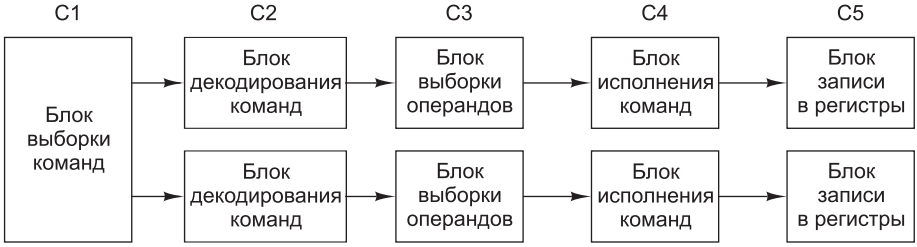


Рис. 2.4. Сдвоенный пятиступенчатый конвейер с общим блоком выборки команд

Сначала конвейеры (как сдвоенные, так и обычные) использовались только в RISC-компьютерах. У процессора 386 и его предшественников их не было. Конвейеры в процессорах компании Intel появились, только начиная с модели 486[1]. Процессор 486 имел один пятиступенчатый конвейер, а Pentium — два таких конвейера. Похожая схема изображена на рис. 2.4, но разделение функций между второй и третьей ступенями (они назывались декодер 1 и декодер 2) было немного другим. Главный конвейер (**u-конвейер**) мог выполнять произвольные команды. Второй конвейер (**v-конвейер**) мог выполнять только простые команды с целыми числами, а также одну простую команду с плавающей точкой (FXCH).

Проверка совместимости команд для параллельного выполнения осуществляется по жестким правилам. Если команды, входящие в пару, были сложными или несовместимыми, выполнялась только одна из них (в u-конвейере). Оставшаяся вторая команда сопоставлялась со следующей командой. Команды всегда выполнялись по порядку. Специальные компиляторы для процессора Pentium объединяли совместимые команды в пары и могли генерировать программы, выполняющиеся быстрее, чем в предыдущих версиях. Измерения показали, что программы, в которых применяются операции с целыми числами, при той же тактовой частое на Pentium выполняются вдвое быстрее, чем на 486 [Pountain, 1993]. Выигрыш в скорости достигался благодаря второму конвейеру.

Переход к четырем конвейерам возможен, но требует громоздкого аппаратного обеспечения (отметим, что компьютерщики, в отличие от фольклористов, не верят в счастливое число три). Вместо этого используется другой подход. Основная идея — один конвейер с большим количеством функциональных блоков, как показано на рис. 2.5. Intel Core, к примеру, имеет сходную структуру (подробно мы рассмотрим ее в главе 4). В 1987 году для обозначения этого подхода был введен термин **суперскалярная архитектура** [Agerwala and Cocke, 1987]. Однако подобная идея нашла воплощение еще более 40 лет назад в компьютере CDC 6600. Этот компьютер вызывал команду из памяти каждые 100 нс и помещал ее в один из 10 функциональных блоков для параллельного выполнения. Пока команды выполнялись, центральный процессор вызывал следующую команду.

[1] Необходимо отметить, что параллельное функционирование отдельных блоков процессора имело место и в предыдущем микропроцессоре (386). Этот механизм стал прообразом 5-ступенчатого конвейера микропроцессора 486. — *Примеч. науч. ред.*

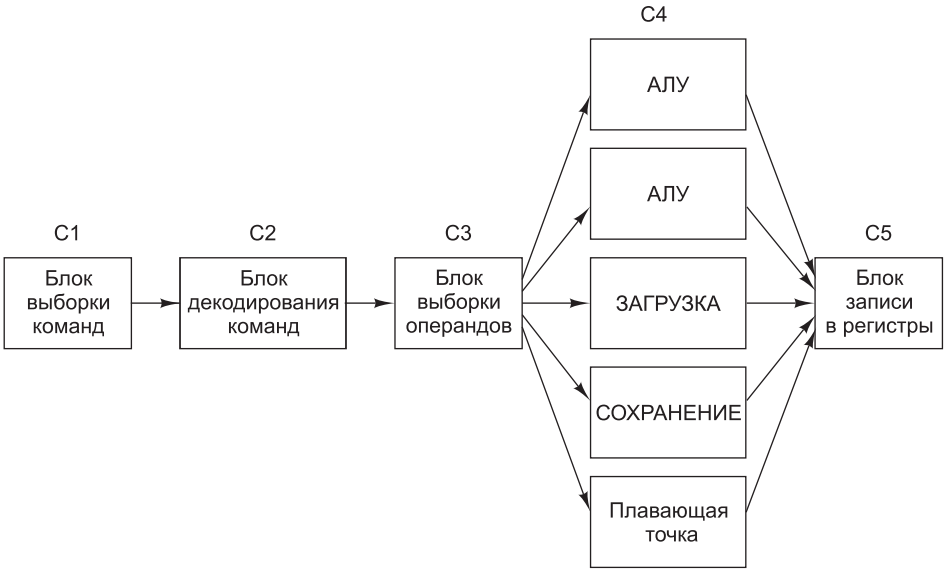


Рис. 2.5. Суперскалярный процессор с пятью функциональными блоками

Со временем определение «суперскалярности» несколько изменилось. Теперь суперскалярными называют процессоры, способные запускать несколько команд (зачастую от четырех до шести) за один тактовый цикл. Естественно, для передачи всех этих команд в суперскалярном процессоре должно быть несколько функциональных блоков. Поскольку в процессорах этого типа, как правило, предусматривается один конвейер, его устройство обычно соответствует рис. 2.5.

В соответствии с этим определением компьютер 6600 формально не был суперскалярным с технической точки зрения — ведь за один тактовый цикл в нем запускалось не больше одной команды. Однако при этом был достигнут аналогичный результат — команды запускались быстрее, чем исполнялись. На самом деле, разница в производительности между ЦП с циклом в 100 нс, передающим за этот период по одной команде четырем функциональным блокам, и ЦП с циклом в 400 нс, запускающим за это время четыре команды, трудноуловима. В обоих процессорах соблюдается принцип превышения скорости запуска над скоростью управления; при этом рабочая нагрузка распределяется между несколькими функциональными блоками.

Отметим, что на выходе ступени 3 команды появляются значительно быстрее, чем ступень 4 способна их обрабатывать. Если бы на выходе ступени 3 команды появлялись каждые 10 нс, а все функциональные блоки делали свою работу также за 10 нс, то на ступени 4 всегда функционировал бы только один блок, что сделало бы саму идею конвейера бессмысленной. В действительности большинству функциональных блоков ступени 4 (точнее, обоим блокам доступа к памяти и блоку выполнения операций с плавающей точкой) для обработки команды требуется значительно больше времени, чем занимает один цикл. Как видно из рис. 2.5, на ступени 4 может быть несколько АЛУ.

Параллелизм на уровне процессоров

Спрос на компьютеры, работающие все с более и более высокой скоростью, не прекращается. Астрономы хотят выяснить, что произошло в первую микросекунду после Большого взрыва, экономисты хотят смоделировать всю мировую экономику, подростки хотят играть в трехмерные интерактивные игры со своими виртуальными друзьями через Интернет. Быстродействие процессоров растет, но у них постоянно возникают проблемы со скоростью передачи информации, поскольку скорость распространения электромагнитных волн в медных проводах и света в оптико-волоконных кабелях по-прежнему остается равной 20 см/нс, независимо от того, насколько умны инженеры компании Intel. Кроме того, чем быстрее работает процессор, тем сильнее он нагревается[1], поэтому возникает задача защиты его от перегрева.

Параллелизм на уровне команд в определенной степени помогает, но конвейеры и суперскалярная архитектура обычно повышают скорость работы всего лишь в 5–10 раз. Чтобы увеличить производительность в 50, 100 и более раз, нужно создавать компьютеры с несколькими процессорами. Ознакомимся с устройством таких компьютеров.

Матричные компьютеры

Многие задачи в физических и технических науках предполагают использование циклов, массивов или других упорядоченных структур. Часто одни и те же вычисления многократно повторяются с разными наборами данных. Упорядоченность и структурированность программ, предназначенных для выполнения такого рода вычислений, очень удобны в плане ускорения вычислений за счет параллельной обработки команд. Существует две схемы ускоренного выполнения больших научных программ: SIMD-процессоры и векторные процессоры. Хотя между этими схемами много общего, как ни парадоксально, первую обычно представляют как параллельный компьютер, а вторую — как добавление расширения параллельного вычислителя.

Компьютеры с распараллеливанием по данным нашли много успешных применений благодаря своей выдающейся эффективности. Они могут обеспечивать существенную вычислительную мощность с меньшим количеством транзисторов по сравнению с альтернативными решениями. Гордон Мур (автор закона Мура) также известен своим замечанием, что кремний стоит около 1 миллиарда долларов за акр (4047 квадратных метров). Таким образом, чем больше вычислительной мощи удастся выжать из этого акра кремния, тем больше денег компьютерная компания заработает на его продаже. Компьютеры с распараллеливанием по данным являются одним из самых эффективных средств для «выжимания» производительности из кремния. Так как все процессоры выполняют одну инструкцию, системе необходим только один «мозг», управляющий компьютером. Соответственно процессору нужна одна ступень выборки команд, одна ступень декодирования и один блок управляющей логики. Так достигается существенная экономия, которая дает параллельным компьютерам большое преимущество

[1] Это не совсем точно. Есть масса примеров, противоречащих этому высказыванию. Тепловыделение, конечно, зависит от частоты переключения элементов, но оно зависит и от размеров этих элементов, и от напряжения, на котором они работают. — *Примеч. науч. ред.*

перед другими процессорами — при условии, что выполняемые программы имеют упорядоченную структуру с большой степенью параллелизма.

SIMD-процессор (Single Instruction-stream Multiple Data-stream — один поток команд с несколькими потоками данных) состоит из большого числа сходных процессоров, которые выполняют одну и ту же последовательность команд применительно к разным наборам данных. Первым в мире таким процессором был ILLIAC IV (университет Иллинойс) [Bouknight et al., 1972]. Первоначально предполагалось сконструировать машину, состоящую из четырех квадрантов, каждый из которых содержал матрицу размером 8 × 8 из блоков процессор/память. Для каждого квадранта имелся один управляющий блок. Он рассылал команды, которые выполнялись всеми процессорами одновременно, при этом каждый процессор использовал собственные данные из собственной памяти. Из-за очень высокой стоимости был построен только один такой квадрант, но он мог выполнять 50 млн операций с плавающей точкой в секунду. Если бы при создании машины использовалось четыре квадранта, она могла бы выполнять 1 млрд операций с плавающей точкой в секунду, и вычислительные возможности такой машины в два раза превышали бы возможности компьютеров всего мира.

Современные графические процессоры (GPU) широко используют SIMD-обработку для обеспечения высокой вычислительной мощности при относительно небольшом количестве транзисторов. Обработка графики отлично подходит для SIMD-процессоров, потому что большинство алгоритмов имеет четкую структуру с повторением операций для пикселов, вершин, текстур и ребер. На рис. 2.6 изображен SIMD-процессор ядра графического процессора Nvidia Fermi. Он содержит до 16 потоковых мультипроцессоров (SM, Stream Multiprocessor) SIMD, каждый из которых содержит 32 процессора SIMD. За каждый цикл планировщик выбирает два потока для выполнения на процессоре SIMD. Затем следующая команда каждого потока выполняется на процессорах SIMD (до 16, хотя при отсутствии достаточного параллелизма данных используется меньшее количество процессоров). Если каждый поток способен выполнить 16 операций за цикл, полностью загруженное ядро графического процессора Fermi с 32 SM будет выполнять целых 512 операций за цикл. Это весьма впечатляющее достижение, если учесть, что четырехъядерный процессор общего назначения того же размера с трудом достигнет 1/32 такой вычислительной мощи.

С точки зрения программиста **векторный процессор** (vector processor) очень похож на SIMD-процессор. Он также чрезвычайно эффективен при выполнении последовательности операций над парами элементов данных. Однако в отличие от SIMD-процессора, все операции сложения выполняются в одном блоке суммирования, который имеет конвейерную структуру. Компания Cray Research, основателем которой был Сеймур Крей, выпустила множество моделей векторных процессоров, начиная с модели Cray-1 (1974).

Оба типа процессоров работают с массивами данных. Оба они выполняют одни и те же команды, которые, например, попарно складывают элементы двух векторов. Однако если у SIMD-процессора столько же суммирующих устройств, сколько элементов в массиве, векторный процессор содержит **векторный регистр**, состоящий из набора традиционных регистров. Эти регистры загружаются из памяти единственной командой, которая фактически делает это последовательно. Команда сложения попарно складывает элементы двух таких векторов, загружая

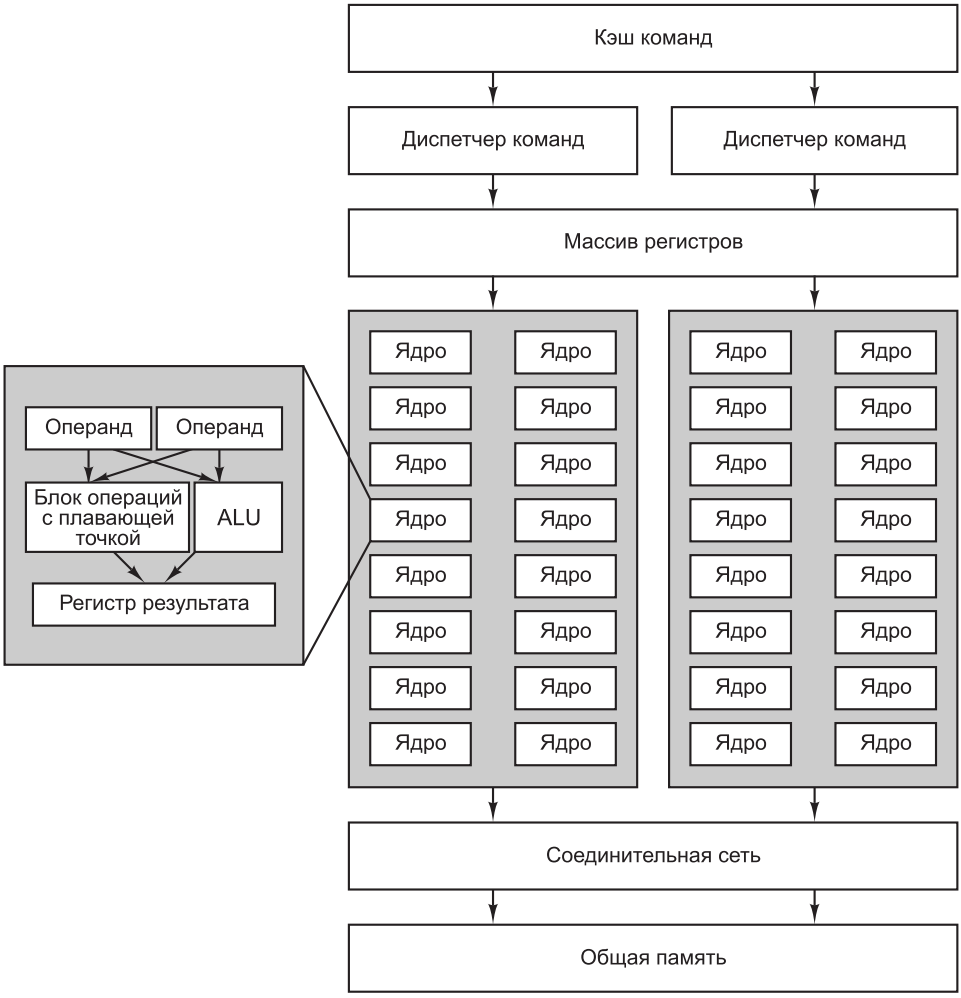


Рис. 2.6. SIMD-ядро графического процессора Fermi

их из двух векторных регистров в суммирующее устройство с конвейерной структурой. В результате из суммирующего устройства выходит другой вектор, который либо помещается в векторный регистр, либо сразу используется в качестве операнда при выполнении другой операции с векторами. Команды SSE (Streaming SIMD Extension) в архитектуре Intel Core используют эту модель расширения для ускорения вычислений с высокой степенью упорядоченности — например, обработки мультимедийных и научных данных. В этом отношении компьютер ILLIAC IV можно считать одним из прародителей процессора Intel Core.

Мультипроцессоры

Элементы процессора, распараллеленного по данным, связаны между собой, поскольку их работу контролирует единый блок управления. Система из нескольких параллельных процессоров, имеющих общую память, называется **мульти-**

процессором. Поскольку каждый процессор может записывать информацию в любую часть памяти и считывать информацию из любой части памяти, чтобы не допустить каких-либо нестыковок, их работа должна согласовываться программным обеспечением. В ситуации, когда два или несколько процессоров имеют возможность тесного взаимодействия, а именно так происходит в случае с мультипроцессорами, эти процессоры называют сильно связанными (tightly coupled).

Возможны разные способы воплощения этой идеи. Самый простой из них — соединение единственной шиной нескольких процессоров и общей памяти. Схема такого мультипроцессора показана на рис. 2.7, *а*.

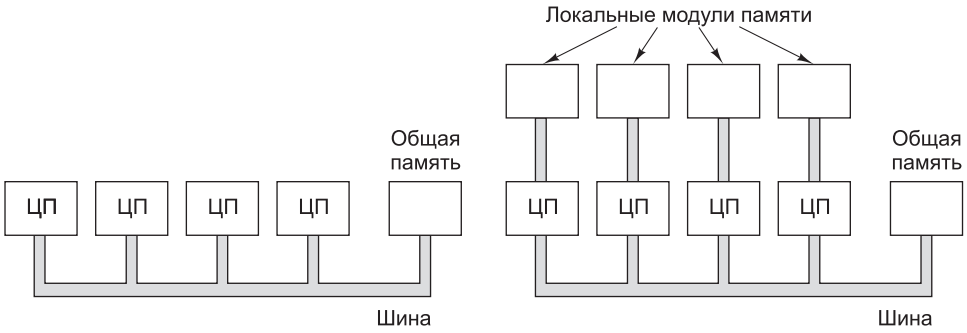


Рис. 2.7. Мультипроцессор с единственной шиной и общей памятью (*а*); мультипроцессор с собственной локальной памятью для каждого процессора (*б*)

Естественно, при наличии большого числа быстродействующих процессоров, которые постоянно пытаются получить доступ к памяти через одну и ту же шину, будут возникать конфликты. Чтобы разрешить эту проблему и повысить производительность компьютера, разработаны различные схемы. Одна из них изображена на рис. 2.7, *б*. В таком компьютере каждый процессор имеет собственную локальную память, недоступную для других процессоров. Эта память используется для тех программ и данных, которые не нужно разделять между несколькими процессорами. При обращении к локальной памяти основная шина не используется, и, таким образом, объем передаваемой по ней информации становится меньше. Возможны и другие варианты решения проблемы (например, кэширование — см. ниже).

Мультипроцессоры имеют преимущество перед другими видами параллельных компьютеров, поскольку с единой общей памятью очень легко работать. Например, представим, что программа ищет раковые клетки на сделанном через микроскоп снимке ткани. Фотография в цифровом виде может храниться в общей памяти, при этом каждый процессор будет обследовать какую-нибудь определенную область фотографии. Поскольку каждый процессор имеет доступ к общей памяти, обследование клетки, расположенной сразу в нескольких областях, не представляет трудностей.

Мультикомпьютеры

Мультипроцессоры с небольшим числом процессоров (≤ 256) разрабатывать достаточно просто, а вот создание больших мультипроцессоров представляет определенные трудности. Сложность заключается в том, чтобы связать все процессоры с общей памятью. Поэтому многие разработчики просто отказались от

идеи разделения памяти и стали создавать системы без общей памяти, состоящие из большого числа взаимосвязанных компьютеров, у каждого из которых имеется собственная память. Такие системы называются **мультикомпьютерами**. В них процессоры являются слабо связанными, в противоположность сильно связанным процессорам в мультипроцессорных системах.

Процессоры мультикомпьютера отправляют друг другу сообщения (что-то вроде электронной почты, но гораздо быстрее). Каждый компьютер не обязательно соединять со всеми другими, поэтому обычно в качестве топологий используются двух- и трехмерные решетки, а также деревья и кольца. Хотя на пути до места назначения сообщения проходят через один или несколько промежуточных компьютеров, время передачи занимает всего несколько микросекунд. Уже работают мультикомпьютеры, содержащие до 250 000 процессоров — например, Blue Gene/P фирмы IBM.

Поскольку мультипроцессоры легче программировать, а мультикомпьютеры — конструировать, появилась идея создания гибридных систем, сочетающих в себе достоинства обеих топологий. Такие компьютеры представляют иллюзию общей памяти, при этом в действительности она не существует. Мы рассмотрим мультипроцессоры и мультикомпьютеры подробнее в главе 8.

Основная память

Память — это тот компонент компьютера, в котором хранятся программы и данные. Также часто встречается термин «запоминающее устройство». Без памяти, откуда процессоры считывают и куда записывают информацию, не было бы современных цифровых компьютеров.

Бит

Основной единицей хранения данных в памяти является двоичный разряд, который называется **битом**. Бит может содержать 0 или 1. Эта самая маленькая единица памяти. (Устройство, в котором хранятся только нули, вряд ли могло быть основой памяти — необходимы, по крайней мере, два значения.)

Часто говорят, что применение двоичной системы счисления в компьютерах объясняется ее «эффективностью». При этом имеется в виду (хотя сами говорящие это редко осознают), что хранение цифровой информации может быть основано на отличиях между разными величинами какой-либо физической характеристики, например напряжения или тока. Чем больше величин нужно различать, тем меньше отличий между смежными величинами и тем менее надежна память. В двоичной системе требуется различать всего две величины, следовательно, это — самый надежный метод кодирования цифровой информации. Если вы не знакомы с двоичной системой счисления, загляните в приложение А.

Считается, что некоторые компьютеры, например мэйнфреймы IBM, используют и десятичную, и двоичную арифметику. На самом деле здесь применяется так называемый **двоично-десятичный код**. Для хранения одного десятичного разряда задействуются 4 бита. Эти 4 бита дают 16 комбинаций для размещения 10 различных значений (от 0 до 9). При этом 6 оставшихся комбинаций не ис-

пользуются. Вот как выглядит число 1944 в двоично-десятичной и в чисто двоичной системах счисления (в обоих случаях используется 16 бит):

- двоично-десятичное представление — 0001 1001 0100 0100;
- двоичное представление — 0000011110011000.

В двоично-десятичном представлении 16 бит достаточно для хранения числа от 0 до 9999, то есть доступно всего 10 000 различных комбинаций, а в двоичном представлении те же 16 бит позволяют получить 65 536 комбинаций. Именно по этой причине говорят, что двоичная система эффективнее.

Однако представим, что могло бы произойти, если бы какой-нибудь гениальный инженер придумал очень надежное электронное устройство, позволяющее хранить разряды от 0 до 9, разделив диапазон напряжения от 0 до 10 вольт на 10 интервалов. Четыре таких устройства могли бы хранить десятичное число от 0 до 9999, то есть 10 000 комбинаций. А если бы те же устройства использовались для хранения двоичных чисел, они могли бы содержать всего 16 комбинаций. Естественно, в этом случае десятичная система была бы более эффективной.

Адреса памяти

Память состоит из **ячеек**, каждая из которых может хранить некоторую порцию информации. Каждая ячейка имеет номер, который называется **адресом**. По адресу программы могут ссылаться на определенную ячейку. Если память содержит n ячеек, они будут иметь адреса от 0 до $n - 1$. Все ячейки памяти содержат одинаковое число битов. Если ячейка состоит из k бит, она может содержать любую из 2^k комбинаций. На рис. 2.8 показаны 3 различных варианта организации 96-разрядной памяти. Отметим, что соседние ячейки по определению имеют последовательные адреса.

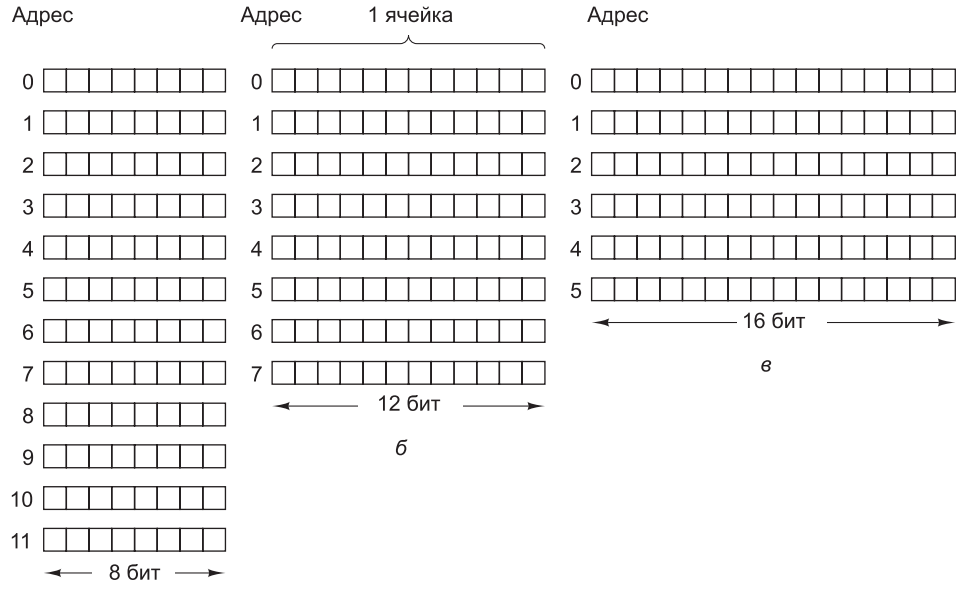


Рис. 2.8. Три варианта организации 96-разрядной памяти

В компьютерах, в которых используется двоичная система счисления (включая восьмеричное и шестнадцатеричное представление двоичных чисел), адреса памяти также выражаются в двоичных числах. Если адрес состоит из m бит, максимальное число адресуемых ячеек составит 2^m. Например, адрес для обращения к памяти, изображенной на рис. 2.8, *а*, должен состоять по крайней мере из 4 бит, чтобы выражать все числа от 0 до 11. При устройстве памяти, показанном на рис. 2.8, *б* и 2.8, *в*, достаточно 3-разрядного адреса. Число битов в адресе определяет максимальное количество адресуемых ячеек памяти и не зависит от числа битов в ячейке. 12-разрядные адреса нужны и памяти из 2^{12} ячеек по 8 бит каждая, и памяти из 2^{12} ячеек по 64 бит каждая.

В табл. 2.1 приведены данные о количестве битов в ячейках памяти некоторых коммерческих компьютеров.

Таблица 2.1. Число битов в ячейке памяти некоторых моделей коммерческих компьютеров

Компьютер	Число битов в ячейке
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

Ячейка — минимальная адресуемая единица памяти. В последние годы практически все производители выпускают компьютеры с 8-разрядными ячейками, которые называются **байтами** (также иногда встречается термин «**октет**»). Байты группируются в **слова**. В компьютере с 32-разрядными словами на каждое слово приходится 4 байт, а в компьютере с 64-разрядными словами — 8 байт. Такая единица, как слово, необходима, поскольку большинство команд производят операции над целыми словами (например, складывают два слова). Таким образом, 32-разрядная машина содержит 32-разрядные регистры и команды для манипуляций с 32-разрядными словами, тогда как 64-разрядная машина имеет 64-разрядные регистры и команды для перемещения, сложения, вычитания и других операций над 64-разрядными словами.

Упорядочение байтов

Байты в слове могут нумероваться слева направо или справа налево. На первый взгляд может показаться, что между этими двумя вариантами нет разницы, но

мы скоро увидим, что выбор имеет большое значение. На рис. 2.9, *а* изображена область памяти 32-разрядного компьютера, в котором байты пронумерованы слева направо (как у компьютеров SPARC или мэйнфреймов IBM). На рис. 2.9, *б* показана аналогичная область памяти 32-разрядного компьютера с нумерацией байтов справа налево (как у компьютеров Intel). В первой из этих систем нумерация начинается с высшего порядка, в связи с чем она относится к категории компьютеров **с прямым порядком следования байтов** (big endian) — в противоположность системам **с обратным порядком следования байтов** (little endian) (рис. 2.9, *б*). Между прочим, эти термины («big endian» и «little endian») заимствованы из «Путешествий Гулливера» Свифта — он, как мы помним, иронизировал по поводу спора политиков о том, с какого конца нужно разбивать яйца. Впервые они были введены в научный оборот в виртуозной статье Коэна (1981).

Рис. 2.9. Память с прямым порядком следования байтов (*а*); память с обратным порядком следования байтов (*б*)

Важно понимать, что в обеих системах 32-разрядное целое число (например, 6) представлено битами 110 в трех крайних правых битах слова, а остальные 29 бит представлены нулями. Если байты нумеруются слева направо, биты 110 находятся в байте 3 (или 7, или 11 и т. д.). Если байты нумеруются справа налево, биты 110 находятся в байте 0 (или 4, или 8 и т. д.). В обеих системах слово, содержащее это целое число, имеет адрес 0.

Если компьютеры содержат только целые числа, никаких сложностей не возникает. Однако многие прикладные задачи требуют использования не только целых чисел, но и цепочек символов и других типов данных. Рассмотрим, например, простую запись данных о персонале, состоящую из строки (имя сотрудника) и двух целых чисел (возраст и номер отдела). Строка завершается одним или несколькими нулевыми байтами, призванными заполнить слово целиком. На рис. 2.10, *а* для записи «Jim Smith, 21 год, отдел 260» ($1 \times 256 + 4 = 260$) представлена схема с нумерацией байтов справа налево, а на рис. 2.10, *б* — с нумерацией байтов слева направо.

Оба эти представления хороши и внутренне непротиворечивы. Проблемы начинаются тогда, когда один из компьютеров пытается переслать запись на другой компьютер по сети. Предположим, что машина с нумерацией байтов слева направо пересылает запись на компьютер с нумерацией байтов справа

Рис. 2.10. Запись данных о сотруднике для машины с прямым порядком следования байтов (*а*); та же запись для машины с обратным порядком следования байтов (*б*); результат передачи записи с машины с прямым порядком следования байтов на машину с обратным порядком следования байтов (*в*); результат перестановки байтов в предыдущем случае (*г*)

налево по одному байту, начиная с байта 0 и заканчивая байтом 19. Для простоты будем считать, что биты не инвертируются при передаче. Таким образом, байт 0 переносится из первой машины на вторую в байт 0 и т. д., как показано на рис. 2.10, *в*.

Компьютер, получивший запись, имя печатает правильно, но возраст получается 21×2^{24}, и номер отдела тоже искажается. Такая ситуация возникает, поскольку при передаче записи порядок следования букв в слове меняется так, как нужно, но при этом порядок следования байтов в целых числах тоже меняется, что приводит к неверному результату.

Очевидное решение проблемы — использование программы, которая бы инвертировала байты в слове после создания копии. Результат такой операции представлен на рис. 2.10, *г*. Мы видим, что числа стали правильными, однако строка превратилась в «MIJTIMS», при этом буква «H» вообще расположилась отдельно. Строка переворачивается потому, что компьютер сначала считывает байт 0 (пробел), затем байт 1 (M) и т. д.

Простого решения не существует. Есть один способ, но он неэффективен. (Нужно перед каждой единицей данных помещать заголовок, информирующий, какой тип данных следует за ним — строка, целое и т. д. Это позволит компьютеру-получателю производить только необходимые преобразования.) Ясно, что отсутствие стандарта упорядочивания байтов является серьезной проблемой при обмене информацией между разными машинами.

Код исправления ошибок

В памяти компьютера из-за всплесков напряжения и по другим причинам время от времени могут возникать ошибки. Для борьбы с ними используются специальные коды, умеющие обнаруживать и исправлять ошибки. В этом случае к каждому слову в памяти особым образом добавляются дополнительные биты. Когда слово считывается из памяти, эти дополнительные биты проверяются, что и позволяет обнаруживать ошибки.

Чтобы понять, как обращаться с ошибками, необходимо внимательно изучить, что представляют собой эти ошибки. Предположим, что слово состоит из m бит данных, к которым мы дополнительно прибавляем r бит (контрольных разрядов). Пусть общая длина слова составит n бит (то есть $n = m + r$). Единицу из n бит, содержащую m бит данных и r контрольных разрядов, часто называют **кодовым словом**.

Для любых двух кодовых слов, например 10001001 и 10110001, можно определить, сколько соответствующих битов в них отличаются. В данном примере таких бита три. Чтобы определить количество отличающихся битов, нужно над двумя кодовыми словами произвести логическую операцию ИСКЛЮЧАЮЩЕЕ ИЛИ и сосчитать число битов со значением 1 в полученном результате. Число битовых позиций, по которым отличаются два слова, называется **интервалом Хэмминга** [Hamming, 1950]. Если интервал Хэмминга для двух слов равен d, значит, достаточно d одноразрядных ошибок, чтобы превратить одно слово в другое. Например, интервал Хэмминга для кодовых слов 11110001 и 00110000 равен 3, поскольку для превращения первого слова во второе достаточно трех одноразрядных ошибок.

Память состоит из m-разрядных слов, следовательно, существуют 2^m вариантов сочетания битов. Кодовые слова состоят из n бит, но из-за способа подсчета контрольных разрядов допустимы только 2^m из 2^n кодовых слов. Если в памяти обнаруживается недопустимое кодовое слово, компьютер знает, что произошла ошибка. При наличии алгоритма подсчета контрольных разрядов можно составить полный список допустимых кодовых слов и из этого списка найти два слова, для которых интервал Хэмминга будет минимальным. Это интервал Хэмминга для полного кода.

Возможности проверки и исправления ошибок определенного кода зависят от его интервала Хэмминга. Для обнаружения d одноразрядных ошибок необходим код с интервалом $d + 1$, поскольку d ошибок не могут превратить одно допустимое кодовое слово в другое допустимое кодовое слово. Соответственно, для исправления d одноразрядных ошибок необходим код с интервалом $2d + 1$, поскольку в этом случае допустимые кодовые слова настолько сильно отличаются друг от друга, что даже если произойдет d изменений, изначальное кодовое слово окажется ближе к ошибочному, чем любое другое кодовое слово, поэтому его без труда можно будет выявить.

В качестве простого примера кода с обнаружением ошибок рассмотрим код, в котором к данным присоединяется один **бит четности**. Бит четности выбирается таким образом, чтобы число битов со значением 1 в кодовом слове было четным (или нечетным). Интервал Хэмминга для этого кода равен 2, поскольку любая одноразрядная ошибка приводит к кодовому слову с неправильной четностью. Другими словами, достаточно двух одноразрядных ошибок для перехода от одного допустимого кодового слова к другому допустимому слову. Такой код может использоваться для обнаружения одиночных ошибок. Если из памяти считывается слово с неверной четностью, поступает сигнал об ошибке. Программа выполняться не сможет, но зато не выдаст неверных результатов.

В качестве простого примера кода исправления ошибок рассмотрим код с четырьмя допустимыми кодовыми словами: 0000000000, 0000011111, 1111100000 и 1111111111.

Интервал Хэмминга для этого кода равен 5. Это значит, что он может исправлять двойные ошибки. Если появляется кодовое слово 0000000111, компьютер знает, что изначально это слово выглядело как 0000011111 (если произошло не более двух ошибок). При появлении трех ошибок (например, слово 0000000000 меняется на 0000000111) этот метод не подходит.

Представим, что мы хотим разработать код с m бит данных и r контрольных разрядов, позволяющий исправлять все одноразрядные ошибки. Каждое из 2^m допустимых слов имеет n недопустимых кодовых слов, которые отличаются от допустимого одним битом. Они образуются инвертированием каждого из n бит в n-разрядном кодовом слове. Следовательно, каждое из 2^m допустимых слов требует $n + 1$ возможных сочетаний битов, приписываемых этому слову (n возможных ошибочных вариантов и один правильный). Поскольку общее число различных сочетаний битов равно 2^n, то $(n + 1)\ 2^m \le 2^n$. Так как $n = m + r$, то $(m + r + 1) \le 2^r$. Эта формула дает нижний предел числа контрольных разрядов, необходимых для исправления одиночных ошибок. В табл. 2.2 показано необходимое количество контрольных разрядов для слов разного размера.

Таблица 2.2. Число контрольных разрядов для кода, способного исправлять одиночные ошибки

Размер исходного слова	Количество контрольных разрядов	Общий размер слова	Процент увеличения слова
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Этого теоретического нижнего предела можно достичь, используя метод Ричарда Хэмминга (1950). Но прежде чем обратиться к указанному алгоритму, давайте рассмотрим простую графическую схему, которая четко иллюстрирует идею кода исправления ошибок для 4-разрядных слов. Диаграмма Венна на рис. 2.11 содержит 3 круга, *A*, *B* и *C*, которые вместе образуют семь секторов. Давайте закодируем в качестве примера слово из 4 бит 1100 в секторы *AB*, *ABC*, *AC* и *BC*, по одному биту в каждом секторе (в алфавитном порядке). Подобное кодирование иллюстрирует рис. 2.11, *а*.

Далее мы добавим бит четности к каждому из трех пустых секторов, чтобы сумма битов в каждом из трех кругов, *A*, *B*, и *C*, получилась четной, как показано на рис. 2.11, *б*. В круге *A* находится 4 числа: 0, 0, 1 и 1, которые в сумме дают четное число 2. В круге *B* находятся числа 1, 1, 0 и 0, которые также при сложении дают четное число 2. Аналогичная ситуация и для круга *C*. В данном примере получилось так, что все суммы одинаковы, но вообще возможны случаи с суммами 0 и 4. Рисунок соответствует кодовому слову, состоящему из 4 бит данных и 3 бит четности.

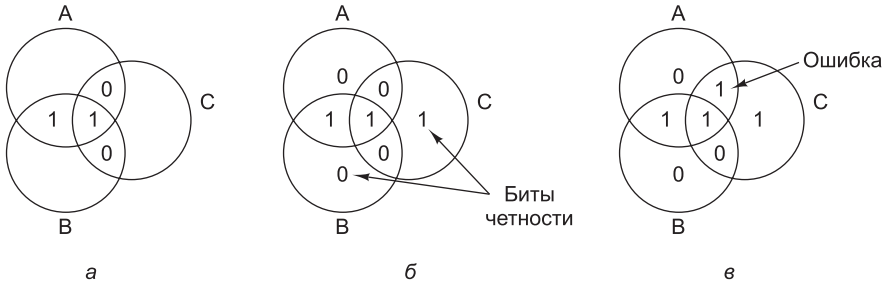


Рис. 2.11. Кодирование числа 1100 (*а*); добавляются биты четности (*б*); ошибка в секторе AC (*в*)

Предположим, что бит в секторе *AC* изменился с 0 на 1, как показано на рис. 2.11, *в*. Компьютер обнаруживает, что круги *A* и *C* являются нечетными. Единственный способ исправить ошибку, изменив только один бит, — возвращение значения 0 биту в секторе *AC*. Таким способом компьютер может исправлять одиночные ошибки автоматически.

А теперь посмотрим, как может использоваться алгоритм Хэмминга при создании кодов исправления ошибок для слов любого размера. В коде Хэмминга к слову, состоящему из m бит, добавляются r бит четности, при этом образуется слово длиной $m + r$ бит. Биты нумеруются с единицы (а не с нуля), причем первым считается крайний левый. Все биты, номера которых — степени двойки, являются битами четности; остальные используются для данных. Например, к 16-разрядному слову нужно добавить 5 бит четности. Биты с номерами 1, 2, 4, 8 и 16 — биты четности, все остальные — биты данных. Всего слово содержит 21 бит (16 бит данных и 5 бит четности). В рассматриваемом примере мы будем использовать проверку на четность (выбор произвольный).

Каждый бит четности позволяет проверять определенные битовые позиции. Общее число битов со значением 1 в проверяемых позициях должно быть четным. Ниже указаны позиции проверки для каждого бита четности:

- бит 1 проверяет биты 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21;
- бит 2 проверяет биты 2, 3, 6, 7, 10, 11, 14, 15, 18, 19;
- бит 4 проверяет биты 4, 5, 6, 7, 12, 13, 14, 15, 20, 21;
- бит 8 проверяет биты 8, 9, 10, 11, 12, 13, 14, 15;
- бит 16 проверяет биты 16, 17, 18, 19, 20, 21.

В общем случае бит b проверяется битами b_1, b_2, ..., b_j, такими что $b_1 + b_2 + ... + b_j = b$. Например, бит 5 проверяется битами 1 и 4, поскольку $1 + 4 = 5$. Бит 6 проверяется битами 2 и 4, поскольку $2 + 4 = 6$ и т. д.

Рисунок 2.12 иллюстрирует построение кода Хэмминга для 16-разрядного слова 1111000010101110. Соответствующим 21-разрядным кодовым словом является 001011100000101101110. Чтобы понять, как происходит исправление ошибок, рассмотрим, что произойдет, если бит 5 изменит значение (например, из-за резкого скачка напряжения). В результате вместо кодового слова 001011100000101101110 получится 001001100000101101110. Будут проверены 5 бит четности. Вот результаты.

- неправильный бит четности 1 (биты 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 содержат пять единиц);
- правильный бит четности 2 (биты 2, 3, 6, 7, 10, 11, 14, 15, 18, 19 содержат шесть единиц);
- неправильный бит четности 4 (биты 4, 5, 6, 7, 12, 13, 14, 15, 20, 21 содержат пять единиц);
- правильный бит четности 8 (биты 8, 9, 10, 11, 12, 13, 14, 15 содержат две единицы);
- правильный бит четности 16 (биты 16, 17, 18, 19, 20, 21 содержат четыре единицы).

Общее число единиц в битах 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 и 21 должно быть четным, поскольку в данном случае используется проверка на четность. Неправильным должен быть один из битов, проверяемых битом четности 1 (а именно 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 и 21). Бит четности 4 тоже неправильный. Это значит, что изменил значение один из следующих битов: 4, 5, 6, 7, 12, 13, 14, 15, 20, 21. Ошибка должна быть в бите, который содержится в обоих списках. В данном случае общими являются биты 5, 7, 13, 15 и 21. Поскольку бит четности 2 правильный, биты 7 и 15 исключаются. Правильность бита четности 8 исключает наличие ошибки в бите 13. Наконец, бит 21 также исключается, поскольку бит четности 16 правильный. В итоге остается бит 5, в котором и кроется ошибка. Поскольку этот бит имеет значение 1, он должен принять значение 0. Именно таким образом исправляются ошибки.

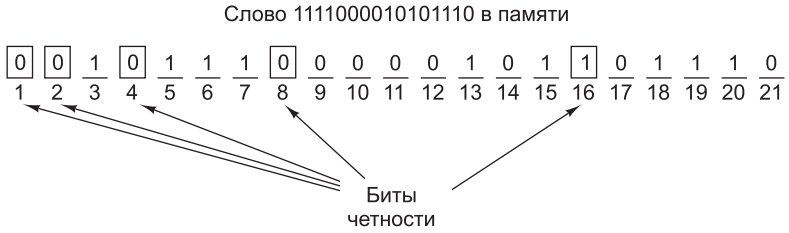


Рис. 2.12. Построение кода Хэмминга для слова 1111000010101110 добавлением к битам данных пяти контрольных разрядов

Чтобы найти неправильный бит, сначала нужно подсчитать все биты четности. Если они правильные, ошибки нет (или есть, но ошибка не однократная). Если обнаружились неправильные биты четности, нужно сложить их номера. Сумма, полученная в результате, даст номер позиции неправильного бита. Например, если биты четности 1 и 4 неправильные, а 2, 8 и 16 правильные, то ошибка произошла в бите 5 (1 + 4).

Кэш-память

Процессоры всегда работали быстрее, чем память. Поскольку процессоры и память совершенствуются параллельно, это несоответствие сохраняется. Поскольку на микросхему можно помещать все больше и больше транзисторов, разработчики процессоров создают конвейерные и суперскалярные архитектуры, что еще больше увеличивает быстродействие процессоров. Разработчики памяти обычно

используют новые технологии для увеличения емкости, а не быстродействия, что делает разрыв еще большим. На практике такое несоответствие в скорости работы приводит к тому, что когда процессор обращается к памяти, проходит несколько машинных циклов, прежде чем он получит запрошенное слово. Чем медленнее работает память, тем дольше процессору приходится ждать, тем больше циклов проходит.

Как мы уже отмечали, есть два пути решения проблемы. Самый простой из них — начать считывать информацию из памяти и при этом продолжать выполнение команд, но если какая-либо команда попытается использовать слово до того, как оно считано из памяти, процессор должен приостановить работу. Чем медленнее работает память, тем чаще будет возникать такая ситуация и тем больше окажется время простоя процессора. Например, если одна команда из пяти обращается к памяти и время обращения составляет 5 циклов, время выполнения увеличивается вдвое по сравнению с мгновенным обращением к памяти. Но если время обращения к памяти составляет 50 циклов, то время выполнения увеличивается уже в 11 раз (5 циклов на выполнение команд + 50 циклов ожидания данных из памяти).

Другое решение проблемы — сконструировать машину, которая не приостанавливает работу, но следит, чтобы программы-компиляторы не использовали слова до того, как они считаны из памяти. Однако это не так просто осуществить на практике. Часто при обработке команды загрузки машина не может выполнять другие действия, поэтому компилятор вынужден вставлять пустые команды, которые не производят никаких операций, но при этом занимают место в памяти. В действительности при таком подходе задержка возникает на уровне программного, а не аппаратного обеспечения, но снижение производительности при этом такое же.

На самом деле эта проблема не технологическая, а экономическая. Инженеры знают, как создать память, которая работает так же быстро, как процессор. Однако ее приходится помещать прямо на микросхему процессора (поскольку информация через шину поступает очень медленно). Размещение памяти большого объема на микросхеме процессора увеличивает его размеры, а следовательно, и стоимость. Впрочем, даже если бы стоимость не имела значения, все равно существуют ограничения на размеры создаваемых процессоров. Таким образом, приходится выбирать между быстрой памятью небольшого объема и медленной памятью большого объема. (Мы, естественно, предпочли бы иметь быструю память большого объема и к тому же дешевую.)

Интересно отметить, что существуют технологии, объединяющие небольшую и быструю память с большой и медленной, что позволяет по разумной цене получить память и с высокой скоростью работы, и большой емкости. Память небольшого объема с высокой скоростью работы называется **кэш-памятью** (от французского слова «cacher» — «прятать»[1]; читается «кашэ»). Далее мы кратко опишем, как используется кэш-память и как она работает. Более подробное описание вы найдете в главе 4.

Основная идея кэш-памяти проста: в ней находятся слова, которые чаще всего используются. Если процессору нужно какое-нибудь слово, сначала он

[1] В английском языке слово «cash» получило значение «наличные (карманные) деньги», то есть то, что под рукой. А уже из него и образовался термин «кэш», который относят к сверхоперативной памяти. — *Примеч. науч. ред.*

обращается к кэш-памяти. Только в том случае, если слова там нет, он обращается к основной памяти. Если значительная часть слов находится в кэш-памяти, среднее время доступа значительно сокращается.

Таким образом, успех или неудача зависит от того, какая часть слов находится в кэш-памяти. Давно известно, что программы не обращаются к памяти наугад. Если программе нужен доступ к адресу A, то скорее всего после этого ей понадобится доступ к адресу, расположенному поблизости от A. Практически все команды обычной программы (за исключением команд перехода и вызова процедур) вызываются из последовательных областей памяти. Кроме того, большую часть времени программа тратит на циклы, когда ограниченный набор команд выполняется снова и снова. Точно так же при работе с матрицами программа, скорее всего, будет многократно обращаться к одной и той же матрице, прежде чем перейдет к чему-либо другому.

Ситуация, когда при последовательных обращениях к памяти в течение некоторого промежутка времени используется только небольшая ее область, называется **принципом локальности**. Этот принцип составляет основу всех систем кэш-памяти. Идея состоит в том, что когда определенное слово вызывается из памяти, оно вместе с соседними словами переносится в кэш-память, что позволяет при очередном запросе быстро обращаться к следующим словам. Общее устройство процессора, кэш-памяти и основной памяти иллюстрирует рис. 2.13. Если слово считывается или записывается k раз, компьютеру требуется сделать одно обращение к медленной основной памяти и $k - 1$ обращений к быстрой кэш-памяти. Чем больше k, тем выше общая производительность.

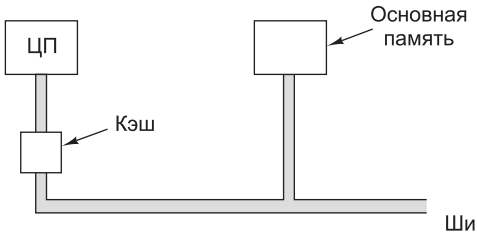


Рис. 2.13. Кэш-память по логике вещей должна находиться между процессором и основной памятью. В действительности существует три возможных варианта размещения кэш-памяти

Мы можем провести и более строгие вычисления. Пусть c — время доступа к кэш-памяти, m — время доступа к основной памяти и h — **коэффициент кэш-попаданий** (hit ratio), который показывает соотношение числа обращений к кэш памяти и общего числа всех обращений к памяти. В нашем примере $h = (k - 1)/k$. Некоторые авторы выделяют **коэффициент кэш-промахов** (miss ratio), равный $1 - h$.

Таким образом, мы можем вычислить среднее время доступа:

$$\textit{Среднее время доступа} = c + (1 - h)\, m.$$

Если $h \to 1$, то есть все обращения делаются только к кэш-памяти, то время доступа стремится к c. С другой стороны, если $h \to 0$, то есть каждый раз нужно обращаться к основной памяти, то время доступа стремится к $c + m$: сначала требуется время c для проверки кэш-памяти (в данном случае безуспешной),

а затем время m для обращения к основной памяти. В некоторых системах обращение к основной памяти может начинаться параллельно с исследованием кэш-памяти, чтобы в случае кэш-промаха цикл обращения к основной памяти уже начался. Однако эта стратегия требует способности останавливать процесс обращения к основной памяти в случае кэш-попадания, что усложняет разработку подобного компьютера.

Основная память и кэш-память делятся на блоки фиксированного размера с учетом принципа локальности. Блоки внутри кэш-памяти обычно называют **строками кэша** (cache lines). При кэш-промахе из основной памяти в кэш-память загружается вся строка, а не только необходимое слово. Например, если строка состоит из 64 байт, обращение к адресу 260 влечет за собой загрузку в кэш-память всей строки (байты с 256 по 319) на случай, если через некоторое время понадобятся другие слова из этой строки. Такой путь обращения к памяти более эффективен, чем вызов каждого слова по отдельности, потому что однократный вызов k слов происходит гораздо быстрее, чем вызов одного слова k раз. Кроме того, превышение строками кэша размера слова означает, что их будет меньше, а следовательно, потребуется меньше непроизводительных затрат ресурсов. Наконец, многие компьютеры (даже 32-разрядные) способны передавать 64 и 128 бит параллельно за один цикл шины.

Кэш-память очень важна для высокопроизводительных процессоров. Однако здесь возникает ряд вопросов. Первый вопрос — объем кэш-памяти. Чем больше объем, тем лучше работает память, но тем дороже она стоит. Второй вопрос — размер строки кэша. Кэш-память объемом 16 Кбайт можно разделить на 1024 строки по 16 байт, 2048 строк по 8 байт и т. д. Третий вопрос — механизм организации кэш-памяти, то есть то, как она определяет, какие именно слова находятся в ней в данный момент. Устройство кэш-памяти мы рассмотрим подробно в главе 4.

Четвертый вопрос — должны ли команды и данные находиться вместе в общей кэш-памяти. Проще всего разработать **объединенную кэш-память** (unified cache), в которой будут храниться и данные, и команды. В этом случае вызов команд и данных автоматически уравновешивается. Однако в настоящее время существует тенденция к использованию **разделенной кэш-памяти** (split cache), когда команды хранятся в одной кэш-памяти, а данные — в другой. Такая архитектура также называется **гарвардской** (Harvard architecture), поскольку идея использования отдельной памяти для команд и отдельной памяти для данных впервые воплотилась в компьютере Marc III, который был создан Говардом Айкеном (Howard Aiken) в Гарварде. Современные разработчики пошли по этому пути, поскольку сейчас широко распространены конвейерные архитектуры, а при конвейерной организации обращения и к командам, и к данным (операндам) должны осуществляться одновременно. Разделенная кэш-память позволяет осуществлять параллельный доступ, а общая — нет. К тому же, поскольку команды обычно не меняются во время выполнения программы, содержание кэша команд не приходится записывать обратно в основную память.

Наконец, пятый вопрос — количество блоков кэш-памяти. В настоящее время очень часто кэш-память первого уровня располагается прямо на микросхеме процессора, кэш-память второго уровня — не на самой микросхеме, но в корпусе процессора, а кэш-память третьего уровня — еще дальше от процессора.

Сборка модулей памяти и их типы

Со времен появления полупроводниковой памяти и до начала 90-х годов все микросхемы памяти производились, продавались и устанавливались в виде отдельных микросхем. Эти микросхемы вмещали от 1 Кбит до 1 Мбит информации и выше. В первых персональных компьютерах часто оставлялись пустые разъемы, чтобы покупатель в случае необходимости мог вставить дополнительные микросхемы памяти.

В настоящее время распространен другой подход. Группа микросхем (обычно 8 или 16) монтируется на одну крошечную печатную плату и продается как один блок. Он называется **SIMM** (Single Inline Memory Module — **модуль памяти с односторонним расположением выводов**) или **DIMM** (Dual Inline Memory Module — **модуль памяти с двухсторонним расположением выводов**). На платах SIMM устанавливается один краевой разъем с 72 контактами; при этом скорость передачи данных за один тактовый цикл составляет 32 бит. Модули DIMM, как правило, снабжаются двумя краевыми разъемами (по одному на каждой стороне платы) с 120 контактами; таким образом, общее количество контактов достигает 240, а скорость передачи данных возрастает до 64 бит за цикл. В настоящее время наиболее распространенными являются DDR3 DIMM — третья версия двухскоростных модулей памяти. Типичный модуль DIMM изображен на рис. 2.14.

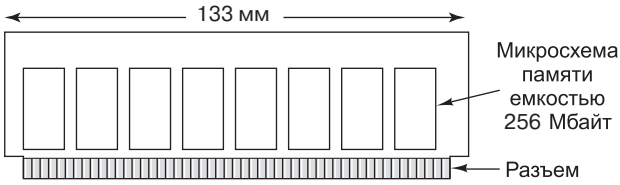


Рис. 2.14. 4-гигабайтный модуль DIMM с восемью 256-мегабайтными микросхемами с каждой стороны. Другая сторона выглядит аналогично.

Обычно модули DIMM содержат 8 микросхем по 256 Мбайт каждая. Таким образом, весь модуль вмещает 2 Гбайт информации. Во многих компьютерах предусматривается возможность установки четырех модулей; следовательно, при использовании модулей по 2 Гбайт общий объем памяти достигает 8 Гбайт (и более при использовании модулей большей емкости).

В портативных компьютерах обычно используется модуль DIMM меньшего размера, который называется **SO-DIMM** (Small Outline DIMM). Модули SIMM и DIMM могут содержать бит четности или код исправления ошибок, однако, поскольку вероятность возникновения ошибок в модуле составляет примерно одну ошибку за 10 лет, в большинстве обычных компьютеров схемы обнаружения и исправления ошибок не применяются.

Вспомогательная память

Каков бы ни был объем основной памяти, ее все равно будет мало. Такова уж наша природа, мы всегда хотим сохранить в памяти компьютера больше данных, чем она может вместить. С развитием технологий людям приходят в голову такие вещи, которые раньше считались совершенно фантастическими. Например, мож-

но вообразить, что Библиотека Конгресса решила представить в цифровой форме и продать полный текст со всеми иллюстрациями всех хранящихся в ней изданий («Все человеческие знания всего за $299,95 доллара»). В среднем каждая книга содержит 1 Мбайт текста и 1 Мбайт упакованных иллюстраций. Таким образом, для размещения 50 млн книг понадобится 10^{14} байт или 100 Тбайт памяти. Для хранения всех существующих художественных фильмов (50 000) необходимо примерно столько же места. Такой объем информации в настоящее время невозможно разместить в основной памяти и вряд удастся это сделать в будущем (по крайней мере, в ближайшие несколько десятилетий).

Иерархическая структура памяти

Иерархическая структура памяти является традиционным решением проблемы хранения больших объемов данных (рис. 2.15). На самом верху иерархии находятся регистры процессора. Доступ к регистрам осуществляется быстрее всего. Дальше идет кэш-память, объем которой сейчас составляет от 32 Кбайт до нескольких мегабайт. Затем следует основная память, объем которой в настоящее время лежит в диапазоне от 1 Гбайт до сотен гигабайт. Затем идут магнитные диски и твердотельные накопители для долгосрочного хранения данных. Нижний уровень иерархии занимают накопители на магнитной ленте и оптические диски для хранения архивов.

Рис. 2.15. Пятиуровневая организация памяти

По мере продвижения сверху вниз по иерархии меняются три параметра. Во-первых, увеличивается время доступа. Доступ к регистрам занимает несколько наносекунд, доступ к кэш-памяти — немного больше, доступ к основной памяти — несколько десятков наносекунд. Дальше идет большой разрыв: доступ к дискам происходит по крайней мере в 10 раз медленнее для твердотельных дисков и в сотни раз медленнее для магнитных дисков. Время доступа к магнитным лентам и оптическим дискам вообще может измеряться в секундах (поскольку эти накопители информации еще нужно взять и поместить в соответствующее устройство).

Во-вторых, растет объем памяти. Регистры могут содержать в лучшем случае 128 байт, кэш-память — десятки мегабайт, основная память — гигабайты, магнитные диски — терабайты. Магнитные ленты и оптические диски хранятся автономно от компьютера, поэтому их совокупный объем ограничивается только финансовыми возможностями владельца.

В третьих, увеличивается количество битов, которое вы получаете за один доллар. Стоимость объема основной памяти измеряется в долларах за мегабайт[1], твердотельных накопителей — в долларах за гигабайт, магнитных дисков и лент — в центах за гигабайт или еще дешевле.

Регистры, кэш-память и основную память мы уже рассмотрели. В следующих разделах мы расскажем о магнитных дисках и твердотельных накопителях, а затем приступим к изучению оптических дисков. Накопители на магнитных лентах мы рассматривать не будем, поскольку используются они редко; к тому же о них практически нечего сказать.

Магнитные диски

Магнитный диск состоит из одной или нескольких алюминиевых поверхностей[2], покрытых магнитным слоем. Изначально их диаметр составлял 50 см, сейчас — от 3 до 9 см, у портативных компьютеров — меньше 3 см, причем это значение продолжает уменьшаться. Головка диска, содержащая индукционную катушку, двигается над поверхностью диска, опираясь на воздушную подушку. Когда через головку проходит положительный или отрицательный ток, он намагничивает поверхность под головкой. При этом магнитные частицы намагничиваются направо или налево в зависимости от полярности тока. Когда головка проходит над намагниченной областью, в ней (в головке) возникает положительный или отрицательный ток, что дает возможность считывать записанные ранее биты. Поскольку диск вращается под головкой, поток битов может записываться, а потом считываться. Конфигурация дорожки диска показана на рис. 2.16.

Дорожкой называется круговая последовательность битов, записанных на диск за его полный оборот. Каждая дорожка делится на **секторы** фиксированной длины. Каждый сектор обычно содержит 512 байт данных. Перед данными располагается **преамбула** (preamble), которая позволяет головке синхронизироваться перед чтением или записью. После данных идет код исправления ошибок (Error-Correcting Code, ECC), в качестве которого используется код Хэмминга или чаще **код Рида–Соломона**, позволяющий исправлять множественные ошибки, а не только одиночные. Между соседними секторами находится **межсекторный интервал**. Многие производители указывают размер неформатированного диска (как будто каждая дорожка содержит только данные), хотя честнее было бы указывать вместимость форматированного диска, на котором не учитываются преамбулы, ECC-коды и межсекторные интервалы. Емкость форматированного диска обычно на 15 % меньше неформатированного.

---

[1] Заметим, что удельная стоимость памяти постоянно снижается, в то время как ее объем — растет. Закон Мура применим и здесь. Сегодня один мегабайт оперативной памяти стоит около 10 центов. — *Примеч. науч. ред.*

[2] В настоящее время компания IBM делает их из стекла. — *Примеч. науч. ред.*

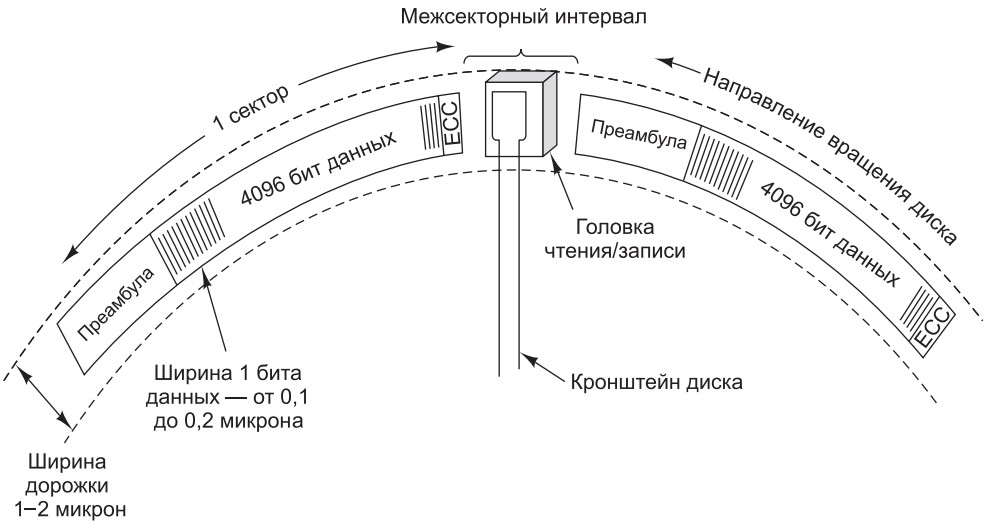


Рис. 2.16. Фрагмент дорожки диска (два сектора)

У всех дисков есть кронштейны, они могут перемещаться туда и обратно по радиусу на разные расстояния от шпинделя, вокруг которого вращается диск. На разных расстояниях от оси записываются разные дорожки. Таким образом, дорожки представляют собой ряд концентрических кругов, расположенных вокруг шпинделя. Ширина дорожки зависит от величины головки и от точности ее перемещения. На сегодняшний момент диски имеют около 50 000 дорожек на сантиметр, то есть ширина каждой дорожки составляет около 200 нанометров (1 нанометр = 1/1 000 000 мм). Следует отметить, что дорожка — это не углубление на поверхности диска, а просто кольцо намагниченного материала, которое отделяется от других дорожек небольшими пограничными областями.

Плотность записи битов на концентрических дорожках отличается от радиальной. Иначе говоря, количество битов на миллиметр, измеряемое вдоль дорожки, отличается от количества битов на миллиметр в направлении от центра диска по радиусу. Плотность записи на дорожке зависит главным образом от качества поверхности диска и чистоты воздуха. Плотность записи современных дисков составляет около 25 гигабит на сантиметр. Радиальная плотность записи зависит от точности позиционирования кронштейна. Таким образом, при записи в радиальном направлении бит занимает существенно больше места, чем в направлении по окружности.

Диски сверхвысокой плотности используют технологию записи, в которой «длинное» измерение битов проходит не вдоль окружности диска, а вертикально — оно как бы уходит в глубь материала. Эти технологии обобщенно называются **перпендикулярной записью**; испытания показали, что они обеспечивают плотность данных до 100 Гбит/см. Весьма вероятно, что эта технология займет ведущее положение в ближайшие годы.

Чтобы достичь высокого качества поверхности и достаточной чистоты воздуха, диски герметично закрываются. Такие диски называются **винчестерами**. Впервые модели, выпущенные фирмой IBM, имели 30 Мбайт фиксированной памяти

и 30 Мбайт сменной памяти. Возможно, эти диски ассоциировались с ружьями «Винчестер» 30–30[1]. В наши дни используется термин «жесткие диски», чтобы устройства не путались с давно вымершими **флопи-дисками** для дискет, использовавшимися на многих первых персональных компьютерах. В компьютерной отрасли трудно подобрать название, которое бы не выглядело смешно 30 лет спустя.

Большинство магнитных дисков состоит из нескольких пластин, расположенных друг под другом, как показано на рис. 2.17. Каждая поверхность снабжена кронштейном и головкой. Кронштейны скреплены таким образом, что одновременно могут перемещаться на разные расстояния от оси. Совокупность дорожек, расположенных на одном расстоянии от центра, называется **цилиндром**. В современных моделях дисков для ПК устанавливается от 1 до 12 пластин, содержащих от 12 до 24 рабочих поверхностей. На одной пластине современных высокопроизводительных дисков может храниться до 1 Тбайт данных, и со временем это значение будет наверняка превышено.

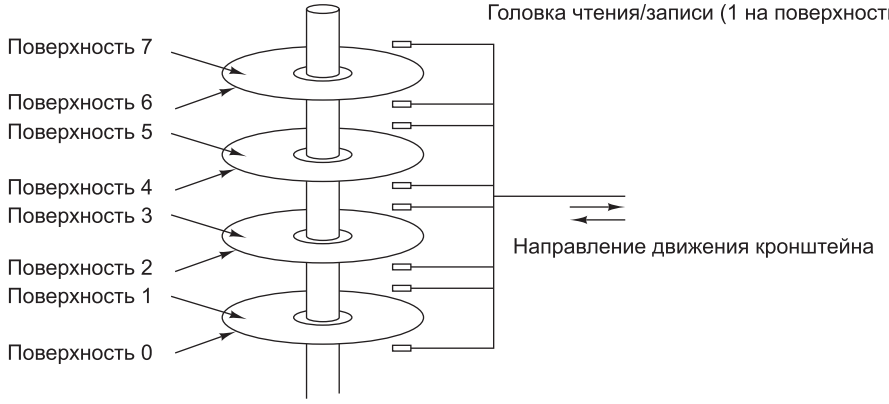


Рис. 2.17. Винчестер с четырьмя дисками

Производительность диска зависит от многих факторов. Чтобы считать или записать сектор, головка должна переместиться на нужное расстояние от оси. Этот процесс называется **позиционированием**. Среднее время поиска между случайно выбранными дорожками составляет от 5 до 10 мс, а для смежных дорожек — менее 1 мс. Когда головка помещается на нужное расстояние от центра, выжидается некоторое время (оно называется **временем ожидания сектора**), пока нужный сектор не окажется под головкой. Большинство дисков вращаются со скоростью 5400, 7200 или 10 800 оборотов в минуту. Таким образом, среднее время ожидания сектора (половина оборота) составляет от 3 до 6 мс. Время передачи информации зависит от плотности записи и скорости вращения. При типичной скорости передачи 150 Мбайт в секунду время передачи одного сектора (512 байт) составляет 3,5 мкс. Следовательно, время передачи информации определяется в основном временем поиска и временем ожидания сектора. Ясно, что считывание случайных секторов из разных частей диска неэффективно.

Следует упомянуть, что из-за наличия преамбул, ECC-кодов, промежутков между секторами, а также из-за того, что определенное время затрачивается на

[1] Двуствольное ружье 30-го калибра. — *Примеч. перев.*

поиск дорожки и ожидание сектора, существует огромная разница между скоростью передачи данных для случаев, когда необходимые данные разбросаны в разных частях диска и когда они находятся в одном месте и считываются последовательно. Максимальная скорость передачи данных в первом случае достигается в тот момент, когда головка располагается над первым битом данных. Однако такая скорость работы может сохраняться только на одном секторе. Для некоторых приложений (например, мультимедийных) имеет значение именно средняя скорость передачи за некоторый период с учетом необходимого времени поиска и времени ожидания сектора.

Немного сообразительности, и старая школьная формула для вычисления длины окружности $c = 2\pi r$ откроет, что суммарная длина внешних дорожек больше, чем длина внутренних. Поскольку все магнитные диски вращаются с постоянной угловой скоростью независимо от того, где находятся головки, возникает очевидная проблема. Раньше при производстве дисков изготовители создавали максимально возможную плотность записи на внутренней дорожке, и при продвижении от центра диска плотность записи постепенно снижалась. Если дорожка содержит, например, 18 секторов, то каждый из них занимает дугу в 20°, и не важно, на каком цилиндре находится эта дорожка.

В настоящее время используется другая стратегия. Цилиндры делятся на зоны (на диске их обычно от 10 до 30). При продвижении от центра диска число секторов на дорожке в каждой зоне возрастает. Это усложняет структуру информации на дорожке, но зато повышает емкость диска, что считается более важным. Все секторы имеют одинаковый размер. Схема диска с пятью зонами изображена на рис. 2.18.

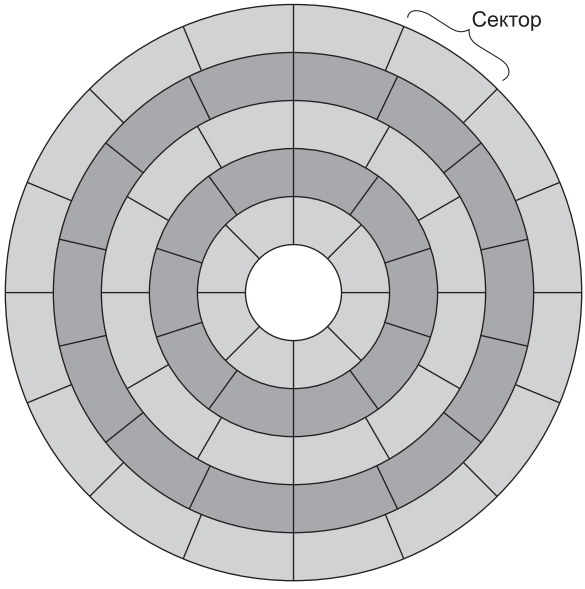


Рис. 2.18. Диск с пятью зонами. Каждая зона содержит несколько дорожек

С диском связан так называемый **контроллер** — микросхема, которая управляет диском. Некоторые контроллеры содержат целый процессор. В задачи

контроллера входит получение от программного обеспечения таких команд, как `READ`, `WRITE` и `FORMAT` (то есть запись всех преамбул), управление перемещением кронштейна, обнаружение и исправление ошибок, преобразование байтов, считываемых из памяти, в непрерывный поток битов, и наоборот. Некоторые контроллеры производят буферизацию и кэширование нескольких секторов на случай их дальнейшего использования, а также пропускают поврежденные секторы. Необходимость последней функции вызвана наличием секторов с поврежденным, то есть постоянно намагниченным, участком. Когда контроллер обнаруживает поврежденный сектор, он заменяет его одним из свободных секторов, которые выделяются специально для этой цели в каждом цилиндре или зоне.

IDE-диски

Прототипом дисков современных персональных компьютеров был диск машины IBM PC XT. Это был диск Seagate на 10 Мбайт, управляемый контроллером Xebec на встроенной карте. У этого диска было 4 головки, 306 цилиндров и по 17 секторов на дорожке. Контроллер мог управлять двумя дисками. Операционная система считывала с диска и записывала на диск информацию. Для этого она передавала параметры в регистры процессора и обращалась с вызовами к системе **BIOS** (Basic Input Output System — **базовая система ввода-вывода**), расположенной во встроенном ПЗУ. Система BIOS выдавала машинные команды для загрузки регистров контроллера, которые начинали передачу данных.

Сначала контроллер помещался на отдельной плате, а с выходом в середине 80-х годов устройств **IDE** (Integrated Drive Electronics — **устройство со встроенным контроллером**) стал встраиваться в материнскую плату[1]. Однако схема с вызовами системы BIOS не изменилась, поскольку необходимо было обеспечить совместимость с более старыми версиями. Обращение к секторам производилось по номерам головки, цилиндра и сектора, причем головки и цилиндры нумеровались с 0, а секторы — с 1. Вероятно, такая ситуация сложилась из-за ошибки одного из программистов BIOS, который писал свой шедевр на ассемблере 8088. Имея 4 бит для номера головки, 6 бит для сектора и 10 бит для цилиндра, диск мог содержать максимум 16 головок, 63 сектора и 1024 цилиндра, то есть всего 1 032 192 сектора. Емкость такого диска составляла 504 Мбайт, и в те времена эта цифра считалась огромной (а вы бы стали сегодня осыпать упреками новую машину, неспособную манипулировать дисками объемом более 1000 Тбайт?).

Вскоре появились диски объемом более 504 Мбайт, но у них была другая геометрия (4 головки, 32 сектора, 2000 цилиндров — это 256 000 секторов). Операционная система не могла обращаться к ним из-за того, что схема вызовов BIOS оставалась неизменной (требование совместимости). В результате контроллеры начали выдавать фиктивную информацию — они «притворялись», что геометрия диска соответствует указанной в BIOS, тогда как в действительности виртуальная геометрия просто накладывалась на реальную. Хотя этот метод работал, он затруднял работу операционных систем, которые размещали данные на диске определенным образом для сокращения времени поиска.

---

[1] Встраиваться он стал в сам винчестер, то есть в печатную плату, расположенную в корпусе винчестера. На материнской плате размещается вторая часть контроллера этого интерфейса. — *Примеч. науч. ред.*

В конце концов на смену IDE-дискам пришли устройства **EIDE** (Extended IDE — **усовершенствованные устройства со встроенным контроллером**), поддерживающие дополнительную схему адресации **LBA (Logical Block Addressing** — линейная адресация блоков). При линейной адресации секторы просто нумеруются от 0 до $2^{28} - 1$. Хотя контроллеру приходится преобразовывать LBA-адреса в адреса головки, сектора и цилиндра, зато объем диска может превышать 504 Мбайт. Однако, к сожалению, в результате родилось новое ограничение на уровне $2^{28} \times 2^9$ байт (128 Гбайт). В 1994 году, когда принимался стандарт EIDE, никому и в голову не приходило, что через некоторое время появятся диски такой емкости. Вообще, комитеты по стандартизации, подобно политикам, зачастую предпочитают откладывать решение проблем, оставляя их своим преемникам.

EIDE-диски и контроллеры имеют и другие усовершенствования. Например, они способны контролировать 4 диска (за счет двух каналов, к каждому из которых можно подключить первичный и вторичный диски), у них более высокая скорость передачи данных (16,67 вместо 4 Мбайт/с), они могут управлять приводами CD-ROM и DVD.

Стандарт EIDE совершенствовался вместе с развитием технологического прогресса, но тем не менее его преемника назвали **ATA-3** (AT Attachment), что выглядело как намек на системы IBM PC/AT (сокращение AT, образованное от словосочетания Advanced Technology — «прогрессивная технология», в этом контексте относилось к прогрессивному на тот момент 16-разрядному процессору с тактовой частотой 8 МГц.) Следующая версия стандарта, названная **ATAPI-4** (ATA Packet Interface — пакетный интерфейс ATA), отличалась скоростью 33 Мбайт/с. В версии ATAPI-5 она достигла 66 Мбайт/с.

Поскольку ограничение в 128 Гбайт, установленное 28-разрядными линейными адресами, становилось все более болезненным, в стандарте ATAPI-6 размер LBA-адреса был увеличен до 48 бит. Лимит этого стандарта — $2^{48} \times 2^9$ (128 Пбайт). Если емкость дисков будет ежегодно возрастать на 50 %, 48-разрядные LBA-адреса останутся актуальными приблизительно до 2035 года. Узнать о том, как решится эта проблема, вы, вероятно, сможете издания эдак из 11-го этой книги. Скорее всего, следующим шагом будет увеличение размера LBA-адреса до 64 бит. В стандарте ATAPI-6 скорость передачи данных удалось довести до 100 Мбайт/с. Кроме того, впервые было уделено внимание проблеме шума.

Настоящий прорыв был совершен в стандарте ATAPI-7. Вместо расширения разъема диска (и, соответственно, скорости передачи данных) появилась спецификация последовательного интерфейса ATA (Serial ATA, **SATA**), позволившего передавать через 7-контактный разъем информацию на скоростях от 150 Мбайт/с (со временем скорость увеличится до 1,5 Гбайт/с). Замена 80-проводного плоского кабеля круглым кабелем диаметром в несколько миллиметров улучшила вентиляцию системного блока. Кроме того, при отправке сигналов через интерфейс SATA используется всего 0,5 В (в сравнении с 5 В по стандарту ATAPI-6), вследствие чего уменьшается общий уровень энергопотребления. Скорее всего, в течение нескольких лет на стандарт SATA будут переведены все компьютеры. В пользу этого варианта развития событий говорит тот факт, что проблема энергопотребления становится все более актуальной — как для информационных центров, которые оснащаются мощными дисковыми фермами, так и для ноутбуков с ограниченными по емкости источниками питания [Gurumurthi et al., 2003].

SCSI-диски

SCSI-диски с точки зрения расположения цилиндров, дорожек и секторов не отличаются от IDE-дисков, но они имеют другой интерфейс и более высокую скорость передачи данных. В 1979 году компания изобретателя дискеты Говарда Шугарта (Hovard Shugart) выпустила диск с интерфейсом SASI (Shugart Associates System Interface). В 1986 году институт ANSI после длительных обсуждений несколько преобразовал этот интерфейс и изменил его название на **SCSI** (Small Computer System Interface — **интерфейс малых вычислительных систем**). Аббревиатура SCSI произносится как «скази». Версии, работающие с более высокой скоростью, получили названия Fast SCSI (10 МГц), Ultra SCSI (20 МГц), Ultra2 SCSI (40 МГц), Ultra3 SCSI (80 МГц) и Ultra4 SCSI (160 МГц). Каждая из этих разновидностей также имела 16-разрядную версию. Собственно, все последние разновидности существуют только в 16-разрядных версиях. Основные параметры всех этих версий сведены в табл. 2.3.

Таблица 2.3. Некоторые допустимые параметры SCSI

Название	Количество разрядов	Частота шины, МГц	Скорость передачи, Мбайт/с
SCSI-1	8	5	5
Fast SCSI	8	10	10
Wide Fast SCSI	16	10	20
Ultra SCSI	8	20	20
Wide Ultra SCSI	16	20	40
Ultra2 SCSI	8	40	40
Wide Ultra2 SCSI	16	40	80
Wide Ultra3 SCSI	16	80	160
Wide Ultra4 SCSI	16	160	320
Wide Ultra5 SCSI	16	320	640

Поскольку у SCSI-дисков высокая скорость передачи данных, они используются во многих высокопроизводительных рабочих станциях и серверах, особенно в конфигурациях RAID (см. ниже).

SCSI — это не просто интерфейс жесткого диска. Это шина, к которой могут подсоединяться SCSI-контроллер и до семи дополнительных устройств. Ими могут быть один или несколько жестких SCSI-дисков, дисководы CD-ROM, сканеры, накопители на магнитной ленте и другие периферийные устройства. Каждое устройство имеет свой идентификационный код от 0 до 7 (до 15 для 16-разрядных версий). У каждого устройства есть два разъема: один — входной, другой — выходной. Кабели соединяют выходной разъем одного устройства с входным разъемом следующего устройства. Это похоже на соединение лампочек в елочной гирлянде. Последнее устройство в цепи должно быть терминальным, чтобы отражения от концов шины не искажали данные в шине. Обычно контроллер помещается на встроенной карте и является первым звеном цепи, хотя это не обязательно.

Самый обычный кабель для 8-разрядного SCSI-устройства имеет 50 проводов, 25 из которых (заземление) спарены с 25 другими, за счет чего обеспечивается хорошая помехоустойчивость, которая необходима для высокой скорости работы. Из 25 проводов 8 используются для данных, 1 — для контроля четности, 9 — для управления, а оставшиеся зарезервированы для будущего применения. 16- и 32-разрядным устройствам требуется еще один кабель для дополнительных сигналов. Кабели могут иметь несколько метров в длину, чтобы обеспечивать связь с внешними устройствами (сканерами и т. п.).

SCSI-контроллеры и периферийные SCSI-устройства могут быть источниками или приемниками команд. Обычно контроллер, действующий как источник, посылает команды дискам и другим периферийным устройствам, которые, в свою очередь, являются приемниками. Команды представляют собой блоки до 16 байтов, которые сообщают приемнику, что нужно делать. Команды и ответы на них оформляются в виде фраз, при этом используются различные сигналы контроля для разграничения фраз и разрешения конфликтных ситуаций, которые возникают, если несколько устройств одновременно пытаются использовать шину. Это очень важно, так как интерфейс SCSI позволяет всем устройствам работать одновременно, что значительно повышает производительность среды, поскольку активизируются сразу несколько процессов. В системах IDE и EIDE активным может быть только одно устройство.

RAID-массивы

Производительность процессоров за последнее десятилетие значительно возросла, увеличиваясь почти вдвое каждые 1,5 года. Однако с производительностью дисков дело обстоит иначе. В 70-х годах среднее время поиска в мини-компьютерах составляло от 50 до 100 миллисекунд. Сейчас время поиска составляет около 10 миллисекунд. Во многих технических отраслях (например, в автомобильной или авиационной промышленности) повышение производительности в 5 или 10 раз за два десятилетия считалось бы грандиозным, но в компьютерной индустрии эти цифры вызывают недоумение. Таким образом, разрыв между производительностью процессоров и дисков все это время продолжал расти.

Как мы уже видели, для того чтобы увеличить быстродействие процессора, используются технологии параллельной обработки данных. Уже на протяжении многих лет разным людям приходит в голову мысль, что было бы неплохо сделать так, чтобы устройства ввода-вывода также могли работать параллельно. В 1988 году Паттерсон в своей статье предложил 6 разных вариантов организации дисковой памяти, которые могли использоваться для повышения производительности, надежности или того и другого [Patterson et al., 1988]. Эти идеи были сразу заимствованы производителями компьютеров, что привело к появлению нового класса устройств ввода-вывода под названием **RAID**. Изначально аббревиатура RAID расшифровывалась как Redundant Array of Inexpensive Disks (избыточный массив недорогих дисков), но позже буква I в аббревиатуре вместо изначального Inexpensive (недорогой) стала означать Independent (независимый) (может, чтобы производители могли делать диски неоправданно дорогими?) RAID-массиву противопоставлялся диск **SLED** (Single Large Expensive Disk — один большой дорогостоящий диск).

Основная идея RAID состоит в следующем. Рядом с компьютером (обычно большим сервером) устанавливается бокс с дисками, контроллер диска замещается RAID-контроллером, данные копируются в RAID-массив, а затем производятся обычные действия. Иными словами, операционная система воспринимает RAID как SLED, при этом у RAID-массива выше производительность и надежность. Поскольку SCSI-диски обладают высокой производительностью при довольно низкой цене и при этом один контроллер может управлять несколькими дисками (до 7 дисков на 8-разрядных моделях SCSI и до 15 на 16-разрядных), большинство RAID-устройств состоит из SCSI-контроллера, предназначенного для управления RAID-масиивом, и бокса SCSI-дисков, которые операционная система воспринимает как один большой диск. Таким образом, чтобы использовать RAID-массив, не требуется никаких изменений в программном обеспечении, что очень выгодно для многих системных администраторов.

RAID-системы имеют несколько достоинств. Во-первых, как уже отмечалось, программное обеспечение воспринимает RAID-массив как один большой диск. Во-вторых, данные на всех дисках RAID-массива распределены по дискам таким образом, чтобы можно было осуществлять параллельные операции. Несколько различных вариантов распределения данных, предложенных Паттерсоном, сейчас известны как RAID-массив уровня 0, RAID-массив уровня 1 и т. д., вплоть до RAID-массива уровня 5. Кроме того, существует еще несколько уровней, которые мы не будем обсуждать. Термин «уровень» выбран неудачно, поскольку здесь нет никакой иерархической структуры. Просто существуют 6 разных вариантов организации дисков с разными характеристиками надежности и производительности.

RAID-массив уровня 0 показан на рис. 2.19, *а*. Он представляет собой виртуальный диск, разделенный на полосы (strips) по k секторов каждая, при этом секторы с 0 по $k - 1$ занимают полосу 0, секторы с k по $2k - 1$ — полосу 1 и т. д. Для $k = 1$ каждая полоса — это сектор, для $k = 2$ каждая полоса — это два сектора и т. д. В RAID-массиве уровня 0 полосы последовательно записываются по кругу, как показано на рис. 2.19, *а*. Это называется **распределением данных** (striping) по дискам. На рисунке изображен RAID-массив с четырьмя дисками. Например, если программное обеспечение выдает команду для считывания блока данных, состоящего из четырех последовательных полос и начинающегося на границе между полосами, то RAID-контроллер разбивает эту команду на 4 отдельные команды, каждую для одного из четырех дисков, и выполняет их параллельно. Таким образом, мы получаем устройство параллельного ввода-вывода без изменения программного обеспечения.

RAID-массив уровня 0 лучше всего работает с большими запросами — чем больше запрос, тем лучше. Если в запросе требуется задействовать полос больше, чем дисков в RAID-массиве, то некоторые диски получают по несколько запросов, и как только такой диск завершает выполнение первого запроса, он приступает к следующему. Задача контроллера состоит в том, чтобы разделить запрос должным образом, послать нужные команды соответствующим дискам в правильной последовательности, а затем правильно записать результаты в память. Производительность при таком подходе очень высокая, и реализация достаточно проста.

RAID-массив уровня 0 хуже всего работает с операционными системами, которые время от времени запрашивают небольшие порции данных (по одному сектору

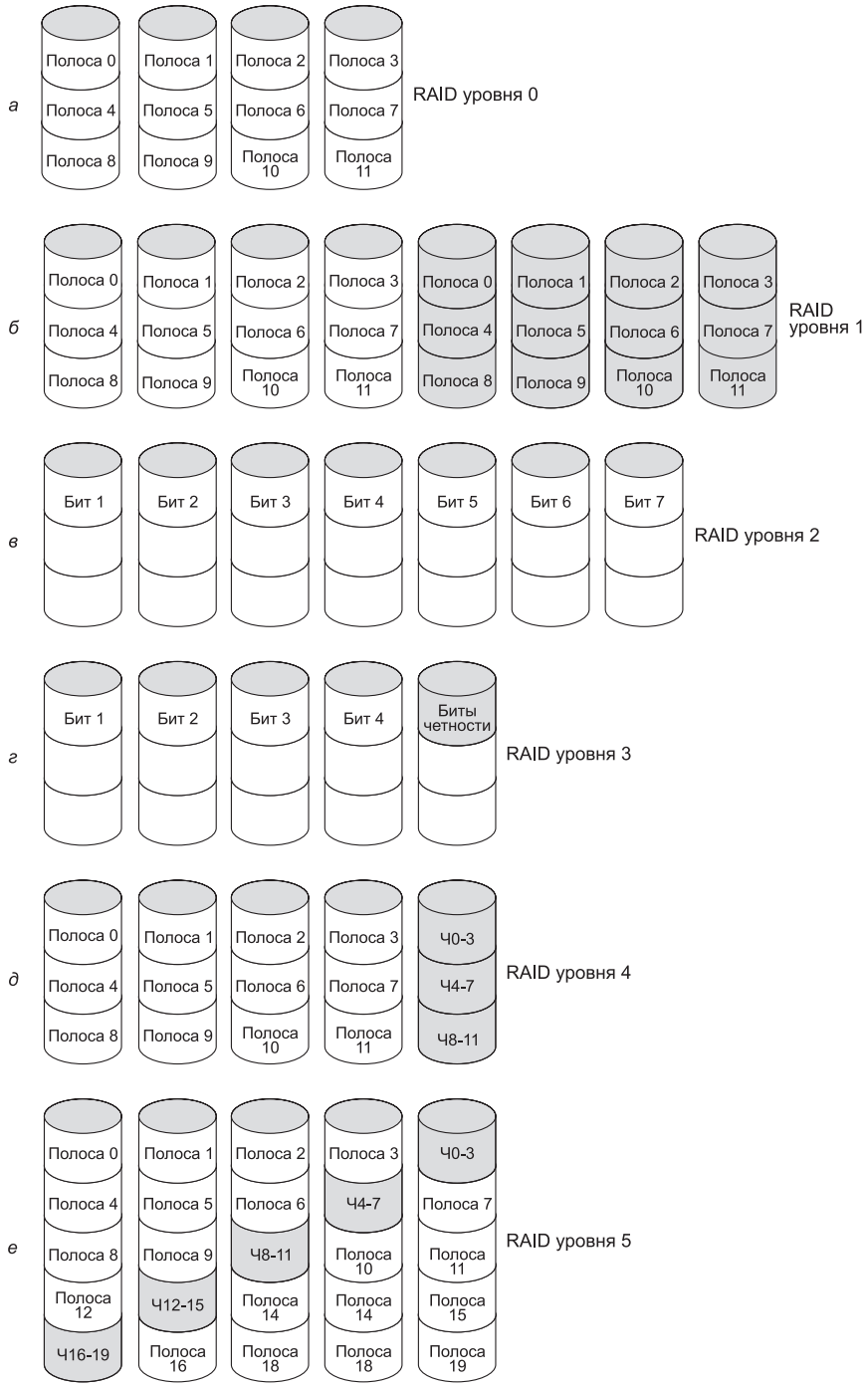


Рис. 2.19. RAID-массивы с нулевого по пятый уровень. Резервные копии и диски четности закрашены серым цветом

за обращение). В этом случае результаты окажутся, конечно, правильными, но не будет никакого параллелизма и, следовательно, никакого выигрыша в производительности. Другой недостаток такой структуры состоит в том, что надежность у нее потенциально ниже, чем у SLED-диска. Например, рассмотрим RAID-массив, состоящий из четырех дисков, на каждом из которых могут происходить сбои в среднем каждые 20 000 часов. То есть сбои в таком RAID-массиве будут случаться примерно через каждые 5000 часов, при этом все данные могут быть утеряны. У SLED-диска сбои происходят также в среднем каждые 20 000 часов, но так как это всего один диск, его надежность в 4 раза выше. Поскольку в описанной разработке нет никакой избыточности, это не «настоящий» RAID-массив.

Следующая разновидность — RAID-массив уровня 1. Он показан на рис. 2.19, *б* и, в отличие от RAID-массива уровня 0, является настоящим RAID-массивом. В этой структуре дублируют все диски, таким образом, получается 4 исходных диска и 4 резервные копии. При записи информации каждая полоса записывается дважды. При считывании может использоваться любая из двух копий, при этом одновременно может происходить загрузка информации с большего количества дисков, чем в RAID-массиве уровня 0. Следовательно, производительность при записи будет такая же, как у обычного диска, а при считывании — гораздо выше (максимум в два раза). Отказоустойчивость отличная: если происходит сбой на диске, вместо него используется копия. Восстановление состоит просто в установке нового диска и копировании всей информации с резервной копии на него.

В отличие от уровней 0 и 1, которые работают с полосами секторов, RAID-массив уровня 2 оперирует словами, а иногда даже байтами. Представим, что каждый байт виртуального диска разбивается на два фрагмента по 4 бита, затем к каждому из них добавляется код Хэмминга, и таким образом получается слово из 7 бит, у которого 1, 2 и 4 — биты четности. Затем представим, что 7 дисков, изображенные на рис. 2.19, *в*, синхронизированы по позиции кронштейна и позиции вращения. Тогда за одну операцию можно записать слово из 7 бит с кодом Хэмминга на 7 дисков, по 1 биту на диск.

Подобная схема использовалась в CM-2 фирмы Thinking Machines. К 32-разрядному слову с данными добавлялось 6 бит четности (код Хэмминга). В результате получалось 38-разрядное кодовое слово, к которому добавлялся дополнительный бит четности, и это слово записывалось на 39 дисков. Общая производительность была огромной, так как одновременно могло записываться 32 сектора данных. При утрате одного из дисков проблем также не возникало, поскольку потеря одного диска означала потерю одного бита в каждом 39-разрядном слове, а с этим код Хэмминга справлялся моментально.

Однако подобная схема требует, чтобы все диски были синхронизированы по вращению. Кроме того, ее имеет смысл использовать, только если имеется достаточно большое количество дисков (даже при наличии 32 дисков для данных и 6 дисков для битов четности накладные расходы составляют 19 %). К тому же имеет место большая нагрузка на контроллер, поскольку он должен вычислять контрольную сумму кода Хэмминга при передаче каждого бита.

RAID-массив уровня 3 представляет собой упрощенную версию RAID-массива уровня 2. Он изображен на рис. 2.19, *г*. Здесь для каждого слова данных вычисляется один бит четности и записывается на диск четности. Как и в RAID-массиве

уровня 2, диски должны быть точно синхронизированы, поскольку каждое слово данных распределено по нескольким дискам.

На первый взгляд может показаться, что один бит четности позволяет только обнаруживать, но не исправлять ошибки. Если речь идет о произвольных ошибках, это наблюдение верно. Однако если речь идет о сбое диска, бит четности обеспечивает исправление ошибки в одном бите, поскольку позиция неправильного бита известна. Если происходит сбой, контроллер выдает информацию, что все биты равны 0. Если в слове возникает ошибка четности, бит с диска, на котором произошел сбой, должен быть равен 1, и, следовательно, он исправляется. Хотя RAID-массивы уровней 2 и 3 обеспечивают очень высокую скорость передачи данных, число запросов от устройств ввода-вывода в секунду не больше, чем при наличии одного диска.

RAID-массивы уровней 4 и 5, как и RAID-массивы начальных уровней, работают с полосами, а не со словами, имеющими биты четности, и не требуют синхронизации дисков. RAID-массив уровня 4 (см. рис. 2.19, *д*) устроен так же, как RAID-массив уровня 0 с тем отличием, что у RAID-массива уровня 4 есть дополнительный диск, на который записываются полосы четности. Например, пусть каждая полоса состоит из k байт. Все полосы объединяются операцией ИСКЛЮЧАЮЩЕГО ИЛИ, и полоса четности для проверки этого отношения также состоит из k байт. Если происходит сбой на диске, утраченные байты могут быть вычислены заново при помощи информации с диска четности.

Такое решение предохраняет от потерь на диске, но значительно снижает производительность в случае небольших исправлений. Если изменяется один сектор, необходимо считать информацию со всех дисков, чтобы опять вычислить биты четности и записать их заново. Вместо этого можно считать с диска прежние данные и прежние биты четности и из них вычислить новые биты четности. Но даже с такой оптимизацией процесса при наличии небольших исправлений требуется произвести два считывания и две записи.

Такие трудности при загрузке данных на диск четности могут быть препятствием для достижения высокой производительности. Эта проблема устраняется в RAID-массиве уровня 5, в котором биты четности распределяются равномерно по всем дискам и записываются по кругу, как показано на рис. 2.19, *е*. Однако в случае сбоя диска восстановить содержание утраченного диска достаточно сложно, хотя и возможно.

Твердотельные накопители

Устройства на базе энергонезависимой флэш-памяти, часто называемые **твердотельными накопителями** или **SSD-дисками** (Solid State Disk), постепенно начинают рассматриваться как высокоскоростная альтернатива традиционным технологиям магнитных дисков. История изобретения SSD — классическое воплощение принципа «Если тебе дают лимон, сделай лимонад». Современная электроника кажется абсолютно надежной, но в действительности транзисторы постепенно изнашиваются в процессе использования. При каждом переключении они ненамного приближаются к состоянию неработоспособности. Один из вероятных путей отказа транзисторов обусловлен эффектом «инжекцией горячих носителей» — механизмом сбоя, при котором в некогда работавший транзистор

внедряется электронный заряд, который навсегда оставляет его во включенном или выключенном состоянии. Хотя обычно такая ситуация рассматривается как смертный приговор для транзистора, Фудзио Масуока (Fujio Masuoka) в ходе своей работы для фирмы Toshiba открыл способ использования этих сбоев для создания новой энергонезависимой памяти. В начале 1980-х годов он изобрел первую флэш-память.

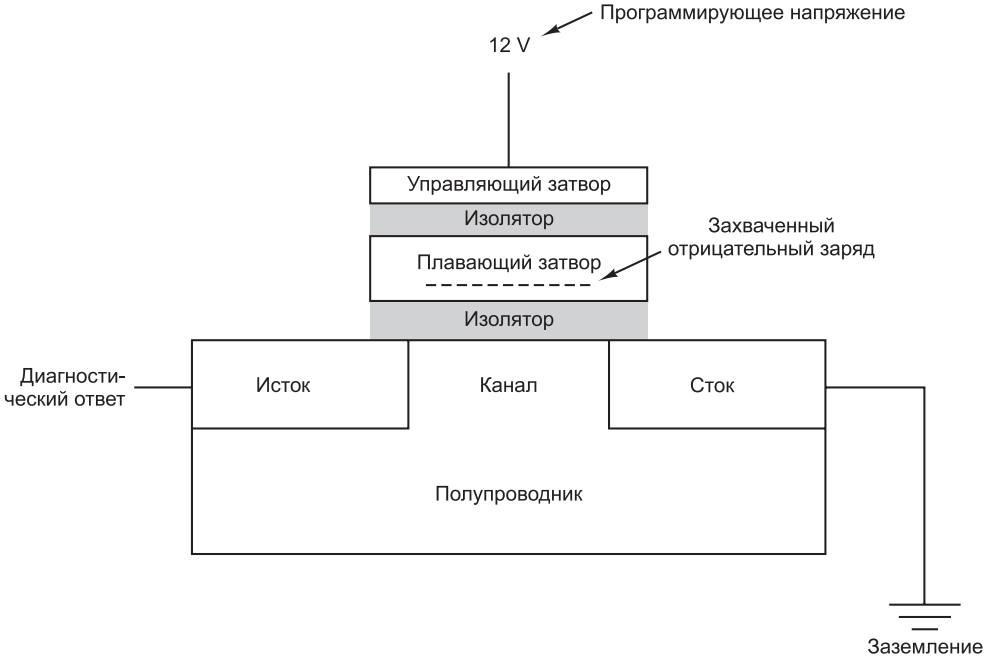


Рис. 2.20. Ячейка флэш-памяти

Флэш-память образуется из множества твердотельных ячеек, состоящих из одного специального флэш-транзистора. Строение ячейки флэш-памяти показано на рис. 2.20. В транзистор встроен плавающий затвор, который может заряжаться и разряжаться при помощи высоких напряжений. До программирования плавающий затвор не влияет на работу транзистора, фактически являясь дополнительным изолятором между управляющим затвором и каналом транзистора. В этом состоянии ячейка ведет себя как простой транзистор.

Для программирования ячейки флэш-памяти на управляющий затвор подается высокое напряжение (в компьютерном мире напряжение в 12 В считается высоким), ускоряющее процесс инжекции горячих электронов в плавающий затвор. Электроны внедряются в плавающий затвор, что приводит к появлению отрицательного заряда внутри транзистора. Внедренный отрицательный заряд увеличивает напряжение, необходимое для включения транзистора; проверяя, включается ли канал при высоком или низком напряжении, можно определить, заряжен ли плавающий затвор, и таким образом получить результат 0 или 1 для ячейки флэш-памяти. Внедренный заряд остается в транзисторе даже при отключении питания, в результате чего ячейка флэш-памяти становится энергонезависимой.

Так как SSD-диски по сути являются памятью, они обладают более высокой производительностью по сравнению с вращающимися магнитными дисками при нулевом времени поиска. Если типичный магнитный диск может обращаться к данным со скоростью 100 Мбит/с, то у SSD эта скорость в два-три раза выше. И поскольку устройство не имеет подвижных частей, оно особенно хорошо подходит для ноутбуков (колебания и перемещения не влияют на его способность обращаться к данным). Недостатком SSD-устройств по сравнению с магнитными дисками является их стоимость. Если память на магнитных дисках обходится в несколько центов за гигабайт, у типичного SSD-диска стоимость гигабайта составляет от одного до трех долларов. Соответственно технология хорошо подходит только для дисков небольшого объема или для ситуаций, в которых расходы не учитываются. Стоимость SSD падает, но эта технология еще не скоро сравняется с дешевыми магнитными дисками. Итак, хотя SSD сегодня заменяют магнитные диски во многих компьютерах, до окончательного вымирания последних еще далеко.

Другой недостаток твердотельных дисков по сравнению с магнитными — их ресурс безотказной работы. Типичная флэш-ячейка перестает функционировать примерно через 100 000 операций перезаписи. Процесс инжекции электронов в плавающий затвор медленно повреждает его и окружающие изоляторы вплоть до окончательной потери работоспособности. Для повышения срока жизни SSD была разработана методика **нивелирования износа**, основанная на распределении записи по всем ячейкам на диске. Каждый раз, когда на устройство записывается новый блок, для записи выбирается блок, относительно давно не использовавшийся. Для этого во флэш-накопителе должна храниться карта блоков — одна из причин, по которой хранение данных на флэш-дисках сопряжено с относительно высокими внутренними затратами. Благодаря нивелированию износа флэш-диск сможет выдержать количество операций записи, равное максимальному количеству операций записи для одной ячейки, умноженному на количество блоков на диске.

Некоторые SSD-диски способны кодировать несколько бит на ячейку, для чего применяются так называемые **многоуровневые флэш-ячейки**. Эта технология основана на тщательном контроле объема заряда, внедряемого в плавающий затвор. Типичные многоуровневые ячейки поддерживают четыре уровня заряда, то есть одна ячейка представляет два бита.

Диски CD-ROM

Оптические диски, которые изначально использовались для записи телевизионных программ, позже стали одними из основных средств хранения информации в компьютерной индустрии. Благодаря большой емкости и низкой цене оптические диски повсеместно применяются для распространения ПО, книг, фильмов и данных других типов, а также для создания архивных копий жестких дисков.

Первые оптические диски были изобретены голландской корпорацией Philips для хранения кинофильмов. Они имели 30 см в диаметре, выпускались под маркой LaserVision, но нигде кроме Японии популярностью не пользовались.

В 1980 году корпорация Philips вместе с Sony разработала компакт-диски (Compact Disc, CD), которые быстро вытеснил виниловые диски, использовав-

шиеся для записи музыки. Описание технических деталей компакт-диска было опубликовано в официальном Международном стандарте (IS 10149), который часто называют **Красной книгой** (по цвету обложки). Международные стандарты издаются Международной организацией по стандартизации (International Organization for Standardization, ISO), которая представляет собой аналог таких национальных организаций стандартизации, как ANSI, DIN и т. п. У каждой такой организации есть свой номер IS (International Standard — международный стандарт). Международный стандарт технических характеристик диска был опубликован для того, чтобы компакт-диски от разных музыкальных издателей и проигрыватели от разных производителей стали совместимыми. Все компакт-диски должны быть 120 мм в диаметре и 1,2 мм в толщину, а диаметр отверстия в середине должен составлять 15 мм. Аудио компакт-диски были первым средством хранения цифровой информации, вышедшим на массовый рынок. Предполагается, что они будут использоваться на протяжении ста лет.

Компакт-диск изготавливается с использованием очень мощного инфракрасного лазера, который выжигает отверстия диаметром 0,8 микрона в специальном стеклянном мастер-диске. По этому мастер-диску делается шаблон с выступами в тех местах, где лазер прожег отверстия. В шаблон вводится жидкая смола (поликарбонат), и, таким образом, получается компакт-диск с тем же набором отверстий, что и в стеклянном диске. На смолу наносится очень тонкий слой алюминия, который, в свою очередь, покрывается защитным лаком. После этого наклеивается этикетка. Углубления в нижнем слое смолы называются **лунками** (pits), а ровные пространства между лунками — **площадками** (lands).

Во время воспроизведения лазерный диод небольшой мощности светит инфракрасным светом с длиной волны 0,78 микрона на сменяющие друг друга лунки и площадки. Лазер находится на той стороне диска, на которую нанесен слой смолы, поэтому лунки для лазера превращаются в выступы на ровной поверхности. Так как лунки имеют высоту в четверть длины световой волны лазера, длина световой волны, отраженной от выступа, составляет половину длины световой волны, отраженной от окружающей выступ ровной поверхности. В результате, если свет отражается от выступа, фотодетектор проигрывателя получает меньше света, чем при отражении от площадки. Именно таким образом проигрыватель отличает лунку от площадки. Хотя, казалось бы, проще всего использовать лунку для записи нуля, а площадку для записи единицы, для единицы надежнее оказалось использовать переход лунка-площадка или площадка-лунка, а отсутствие перехода — для нуля.

Лунки и площадки записываются по спирали. Запись начинается на некотором расстоянии от отверстия в центре диска и продвигается к краю, занимая 32 мм диска. Спираль проходит 22 188 оборотов вокруг диска (примерно 600 на 1 мм). Если спираль распрямить, ее длина составит 5,6 км. Спираль изображена на рис. 2.21.

Чтобы музыка звучала нормально, лунки и площадки должны сменяться с постоянной *линейной* скоростью. Следовательно, скорость вращения компакт-диска должна постепенно снижаться по мере продвижения считывающей головки от центра диска к внешнему краю. Когда головка находится на внутренней стороне диска, то, чтобы достичь желаемой скорости 120 см/с, частота вращения должна составлять 530 оборотов в минуту. Когда головка находится на внешней стороне

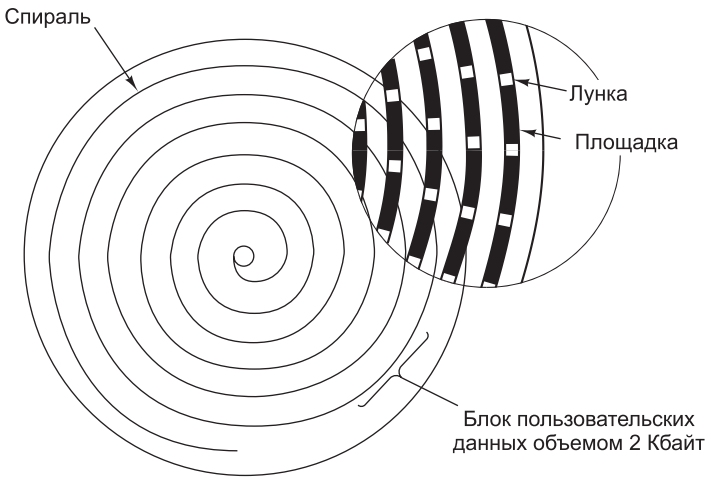


Рис. 2.21. Схема записи компакт-диска

диска, частота вращения падает до 200 оборотов в минуту, что позволяет обеспечить такую же линейную скорость. Этим компакт-диск, вращающийся с постоянной линейной скоростью, отличается от магнитного диска, вращающегося с *постоянной угловой скоростью* независимо от того, где в этот момент находится головка. Кроме того, частота вращения компакт-диска (530 оборотов в минуту) сильно отличается от частоты вращения магнитных дисков, которая составляет от 3600 до 7200 оборотов в минуту.

В 1984 году Philips и Sony осознали потенциал использования компакт-дисков для хранения компьютерных данных. Они опубликовали **Желтую книгу**, в которой определили точный стандарт того, что они назвали CD-ROM (Compact Disc-Read Only Memory — **постоянная память на компакт-диске**). Чтобы выйти на развитый к тому времени рынок аудио компакт-дисков, компьютерные компакт-диски должны были быть такого же размера, как аудиодиски, механически и оптически совместимыми с ними и производиться по той же технологии. Вследствие такого решения потребовались низкоскоростные двигатели, работающие с переменной скоростью, но зато стоимость производства одного компакт-диска составляла менее одного доллара.

В Желтой книге определены форматы компьютерных данных. В ней также описаны усовершенствованные приемы исправления ошибок, что является существенным шагом, поскольку для компьютерщиков, в отличие от любителей музыки, ошибка в одном бите становится критичной. Разметка компакт-диска состоит в кодировании каждого байта 14-разрядным символом. Как уже отмечалось, 14 бит достаточно для того, чтобы закодировать кодом Хэмминга 8-разрядный байт, при этом останется два лишних бита. На самом деле используется более мощная система кодировки. Перевод из 14- в 8-разрядную систему для считывания информации производится аппаратно с помощью поисковых таблиц.

На следующем уровне 42 последовательных символа формируют **фрейм** из 588 бит. Каждый фрейм содержит 192 бита данных (24 байта). Оставшиеся 396 бит используются для исправления ошибок и контроля. У аудио и компьютерных компакт-дисков эта система одинакова.

У компьютерных компакт-дисков каждые 98 фреймов группируются в сектор, как показано на рис. 2.22. Каждый сектор начинается с преамбулы из 16 байт, первые 12 из которых образуют значение 00FFFFFFFFFFFFFFFFFF00 (в шестнадцатеричной системе счисления), по которому проигрыватель распознает начало сектора. Следующие 3 байта содержат номер сектора. Номер необходим, поскольку поиск на компакт-диске, на котором данные записаны по спирали, гораздо сложнее, чем на магнитном диске, где данные записаны на концентрических дорожках. Чтобы найти определенный сектор, программное обеспечение подсчитывает, куда приблизительно нужно направляться; туда помещается считывающая головка, а затем начинается поиск преамбулы, чтобы установить, насколько верен был подсчет. Последний байт преамбулы определяет тип диска.

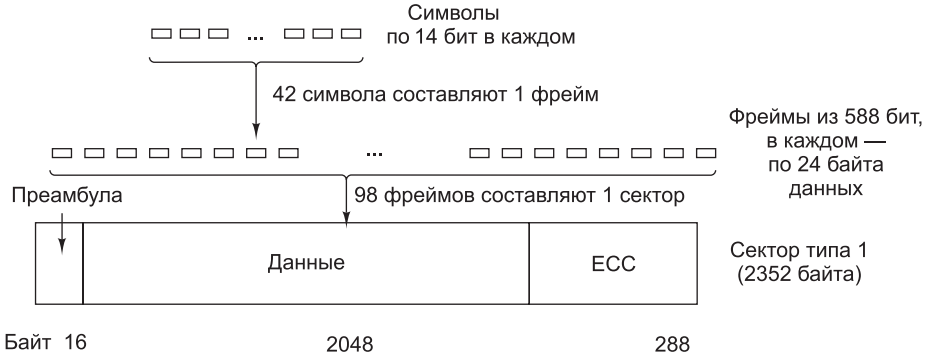


Рис. 2.22. Схема расположения данных на компакт-диске

Желтая книга определяет два типа дисков. На рис. 2.22 показана схема расположения данных для типа 1, где преамбула составляет 16 байт, данные — 2048 байт, а код исправления ошибок — 228 байт (код Рида–Соломона). На дисках типа 2 данные и коды исправления ошибок объединяются в поле данных размером 2336 байт. Такая схема применяется для приложений, которые не нуждаются в исправлении ошибок (а вернее, не могут выделить время для этого), например аудио и видео. Отметим, что для обеспечения высокой степени надежности используются три схемы исправления ошибок: в пределах символа, в пределах фрейма и в пределах сектора. Одиночные ошибки в битах исправляются на самом нижнем уровне, пакеты ошибок — на уровне фреймов, а все остаточные ошибки — на уровне секторов. Для поддержания такой надежности необходимо 98 фреймов по 588 бит (7203 байта), чтобы получить 2048 байт полезной нагрузки. Таким образом, эффективность составляет всего 28 %.

Односкоростные устройства для чтения компакт-дисков считывают 75 секторов в секунду, что обеспечивает скорость передачи данных 153 600 байт/с для дисков типа 1 и 175 200 байт/с для дисков типа 2. Двухскоростные устройства работают в два раза быстрее и т. д., до самой высокой скорости. Стандартный аудио компакт-диск «вмещает» 74 минуты музыки, что соответствует значению 681 984 000 байт, или 650 Мбайт, так как 1 Мбайт = 2^{20} байт (1 048 576 байт), а не 1 000 000 байт.

Как обычно, при появлении новой технологии находятся люди, пытающиеся выйти за границы допустимого. При проектировании стандарта CD-ROM Philips

и Sony проявили осторожность, и процесс записи останавливался задолго до достижения внешней границы диска. По прошествии некоторого времени отдельные производители дисководов позволили своим устройствам выходить за «официальные» рамки, приближаясь к физической границе носителя. Емкость диска при этом увеличивалась с 640 Мбайт до 700. Но с развитием технологии и появлением пустых дисков, изготовленных по улучшенному стандарту, емкость в 703,12 Мбайт (360 000 2048-байтовых секторов вместо 333 000) стала новой нормой.

Отметим, что даже устройство для чтения компакт-дисков со скоростью, обозначаемой как 32x (4 915 200 байт/с), несравнимо по быстродействию с магнитным диском SCSI-2 (10 Мбайт/с), хотя многие устройства для чтения компакт-дисков используют интерфейс SCSI (кроме того, применяется интерфейс IDE). Отсюда ясно, что компакт-диски по производительности значительно уступают магнитным дискам, несмотря на их большую емкость[1].

В 1986 году корпорация Philips опубликовала **Зеленую книгу**, добавив графику и возможность помещать аудио- и видеоданные, а также обычные данные в одном секторе, что было необходимо для мультимедийных компакт-дисков.

Последняя проблема, которую нужно было разрешить при разработке компакт-дисков, — совместимость файловых систем. Чтобы один и тот же компакт-диск можно было использовать на разных компьютерах, необходимо было соглашение по поводу файловой системы для компакт-дисков. Чтобы выпустить такое соглашение, представители разных компьютерных компаний встретились на озере Тахо на границе Калифорнии и Невады и разработали файловую систему, которую они назвали **High Sierra** (по названию населенного пункта, в котором они собрались). Позднее эта система превратилась в международный стандарт (IS 9660). Существуют три уровня этого стандарта. На уровне 1 допустимы имена файлов длиной до 8 символов, за именем файла может следовать расширение до 3 символов (соглашение по наименованию файлов в MS-DOS). Имена файлов могут содержать только буквы в верхнем регистре, цифры и символ подчеркивания. Поддерживается вложение каталогов на глубину не более 8 уровней. Имена каталогов могут не содержать расширения. На первом уровне требуется, чтобы все файлы были смежными, что не представляет особых трудностей в случае с носителем, на который информация записывается только один раз. Любой компакт-диск, который соответствует стандарту IS 9660 уровня 1, может быть прочитан в системе MS-DOS, на компьютерах Apple, Unix и практически любых других. Производители компакт-дисков считают это свойство большим плюсом.

Уровень 2 стандарта IS 9660 допускает имена файлов длиной до 32 символов, а на уровне 3 допускается несмежное расположение файлов. Расширения Rock Ridge (названные так в честь города из фильма «Горящие седла» Мела Брукса) допускают очень длинные имена файлов (для UNIX), идентификаторы UID и GID, а также символические ссылки, однако компакт-диски, не соответствующие уровню 1, не читаются на старых компьютерах.

---

[1] Емкость компакт-дисков на два порядка ниже емкости современных магнитных дисков. — *Примеч. науч. ред.*

Диски CD-R

Вначале оборудование, необходимое для изготовления мастер-дисков (как аудио, так и компьютерных), было очень дорогим. Но, как это обычно происходит в компьютерной промышленности, ничего не остается дорогим слишком долго. К середине 90-х годов записывающие устройства для компакт-дисков размером не больше проигрывателя стали обычными и общедоступными, их можно было приобрести в любом магазине компьютерной техники. Эти устройства все еще отличались от магнитных дисков, поскольку информацию, записанную однажды на компакт-диск, уже нельзя было стереть. Тем не менее они быстро нашли сферу применения в качестве дополнительных носителей информации, а основными носителями продолжали служить жесткие диски. Кроме того, отдельные лица и маленькие компании получили возможность выпускать собственные компакт-диски небольшими партиями или производить мастер-диски и отправлять их на крупные коммерческие предприятия, занимающиеся изготовлением копий. Такие диски называются **CD-R** (CD-Recordable — **записываемый компакт-диск**).

Основой диска CD-R является 120-миллиметровая поликарбонатная заготовка. Такие же заготовки используются при производстве дисков CD-ROM. Однако диски CD-R отличаются от дисков CD-ROM тем, что содержат канавку шириной 0,6 мм, призванную направлять лазер при записи. Для поддержания постоянной обратной связи канавка имеет синусоидальную форму с отклонением 0,3 мм и частотой ровно 22,05 кГц, чтобы можно было точно определить скорость вращения и в случае необходимости отрегулировать ее. CD-R выглядит как обычный диск, только он не серебристого, а золотистого цвета, так как для изготовления отражающего слоя вместо алюминия используется золото. В отличие от обычных компакт-дисков, лунки и площадки на дисках CD-R имитируются путем изменения отражающей способности поверхности. Для этого между слоем поликарбоната и отражающим слоем золота помещается слой красителя (рис. 2.23). Используются

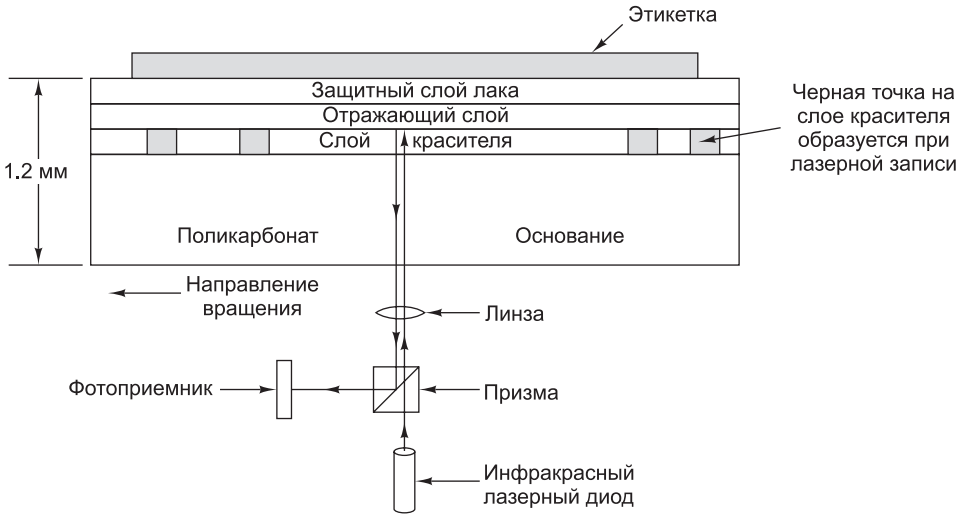


Рис. 2.23. Поперечное сечение диска CD-R и лазера (масштаб не соблюден). Обычный компакт-диск имеет близкую структуру, но у него отсутствует слой красителя, а вместо слоя золота используется слой алюминия с лунками

два вида красителя: цианин зеленого цвета и фталоцианин желтовато-оранжевого цвета. Химики могут спорить до бесконечности, который из них лучше. В дальнейшем золотой отражающий слой был заменен алюминиевым.

На начальной стадии слой красителя прозрачен, что дает возможность свету лазера проходить сквозь него и отражаться от слоя золота. При записи информации мощность лазера увеличивается до 8–16 мВт. Когда луч достигает красителя, краситель нагревается, и в результате разрушается химическая связь. Такое изменение молекулярной структуры создает темное пятно. При чтении (когда мощность лазера составляет 0,5 мВт) фотодетектор улавливает разницу между темными пятнами, где краситель был поврежден, и прозрачными областями, где краситель остался нетронутым. При воспроизведении диска даже на обычном устройстве для считывания компакт-дисков или на аудиопроигрывателе это отличие воспринимается как разница между лунками и площадками традиционного диска CD-ROM.

Ни один новый вид компакт-дисков не обходился без публикации параметров в книге определенного цвета. В случае с CD-R это была **Оранжевая книга**, вышедшая в 1989 году. Этот документ описывает диск CD-R, а также новый формат, **CD-ROM XA**, который позволяет записывать информацию на CD-R постепенно (несколько секторов сегодня, несколько завтра, несколько через месяц и т. д.). Группа последовательных секторов, записываемых за один раз, называется **дорожкой**.

Одним из первых применений формата CD-R стал фото компакт-диск фирмы Kodak. При использовании этой технологии экспонированная пленка и фото компакт-диск со старыми снимками вставляются в проявочную машину, которая выдает компакт-диск обратно, предварительно дописав на него (после старых снимков) снимки с экспонированной пленки. Новый пакет данных, полученный в результате сканирования негативов, записывается на компакт-диск в виде отдельной дорожки. Такой способ записи необходим, поскольку заготовки для дисков CD-R слишком дорого стоят и записывать каждую новую пленку на новый диск невыгодно.

Однако с появлением этой технологии записи возникла новая проблема. До выхода Оранжевой книги в начале всех компакт-дисков находилась единая таблица **VTOC** (Volume Table of Contents — **таблица оглавления диска**), из-за которой дописывать информацию на диск было невозможно. Решением проблемы стало предложение назначать каждой дорожке диска свою таблицу VTOC. В число файлов, перечисленных в таблице VTOC, могут включаться все файлы с предыдущих дорожек или некоторые из них. После того как диск CD-R вставляется в считывающее устройство, операционная система начинает искать среди дорожек самую последнюю таблицу VTOC, которая и отражает текущее состояние диска. Если в таблице VTOC перечислить только некоторые, а не все файлы с имеющихся дорожек, может создаться впечатление, что файлы удалены. Дорожки можно группировать в **сессии**. В этом случае мы говорим о **многосессионных** компакт-дисках. Стандартные аудиопроигрыватели не могут работать с многосессионными компакт-дисками, поскольку пытаются искать единую таблицу VTOC в начале диска.

С появлением дисков CD-R отдельные пользователи и компании получили возможность без труда копировать компьютерные и музыкальные компакт-диски, причем иногда с нарушением авторских прав. Были придуманы разные средства, препятствующие производству пиратской продукции и затрудняющие чтение

компакт-дисков с помощью программного обеспечения, разработанного не производителем данного диска. Один из таких способов — запись на диск заведомо завышенной информации о размере файлов на диске. Это препятствует копированию файлов на жесткий диск с помощью обычного программного обеспечения. Настоящие размеры файлов включаются в специализированное программное обеспечение, предназначенное для чтения данного компакт-диска, или прячутся где-нибудь на компакт-диске (часто в зашифрованном виде). При другом способе в избранные секторы вставляются заведомо неправильные ECC-коды. Программное обеспечение, прилагаемое к данному компакт-диску, исправляет эти ошибки, а обычное программное обеспечение работать не может, поскольку коды заведомо неправильные. Кроме того, возможно использование нестандартных промежутков между дорожками и других физических «дефектов».

Диски CD-RW

Хотя люди привыкли иметь дело с такими носителями информации, перезаписать которые невозможно (как, например, бумагу, фотопленку или виниловую пластинку), спрос на перезаписываемые компакт-диски все равно остается. В настоящее время появилась технология **CD-RW** (CD-ReWritable — **перезаписываемый компакт-диск**), в которой используется носитель такого же размера, как и для дисков CD-R, однако вместо красителя (цианина или фталоцианина) при производстве CD-RW в качестве слоя записи применяется сплав серебра, индия, сурьмы и теллура. Этот сплав имеет два состояния: кристаллическое и аморфное, которые обладают разной отражательной способностью.

Устройства для записи компакт-дисков снабжены лазером с тремя уровнями мощности. При самой высокой мощности лазер расплавляет сплав, меняя его состояние из кристаллического с высокой отражательной способностью в аморфное с низкой отражательной способностью, — так получается лунка. При средней мощности сплав расплавляется и возвращается обратно в естественное кристаллическое состояние, при этом лунка снова превращается в площадку. При низкой мощности лазер определяет состояние материала (обеспечивая считывание информации), никакой смены состояний при этом не происходит.

Диски CD-RW не заменили собой диски CD-R, поскольку заготовки для дисков CD-RW гораздо дороже заготовок CD-R. Кроме того, для приложений, ориентированных на создание резервных копий жестких дисков, большим плюсом является тот факт, что с CD-R нельзя случайно стереть информацию.

DVD-диски

Компакт-диски основных форматов (CD и CD-ROM) использовались с 1980 года. С тех пор технологии продвинулись вперед, оптические диски большой емкости сейчас вполне доступны по цене и пользуются большим спросом. Голливуд с радостью переходит с аналоговых видеокассет на цифровые видеодиски, поскольку они лучше по качеству, их дешевле производить, они дольше служат, занимают меньше места на полках магазинах, их не нужно перематывать. Было очевидно, что колесо прогресса оптических дисков готово сделать новый поворот.

Такое развитие технологий, а также спроса на продукцию трех чрезвычайно богатых и мощных отраслей промышленности, и стало причиной рождения DVD-

дисков. Изначально аббревиатура **DVD** расшифровывалась как Digital Video Disk (**цифровой видеодиск**), сейчас она официально превратилась в Digital Versatile Disk (**цифровой многоцелевой диск**). DVD-диски в целом похожи на компакт-диски. Как и обычные компакт-диски, они имеют 120 мм в диаметре, создаются на основе поликарбоната и содержат лунки и площадки, которые освещаются лазерным диодом и считываются фотодетектором. Однако существует несколько отличий:

- меньший размер лунок (0,4 микрона вместо 0,8 микрона, как у обычного компакт-диска);
- более плотная спираль (0,74 микрона между дорожками вместо 1,6 микрона);
- красный лазер (с длиной волны 0,65 микрона вместо 0,78 микрона).

В совокупности эти усовершенствования дали семикратное увеличение емкости (до 4,7 Гбайт). Считывающее устройство для DVD 1x работает со скоростью 1,4 Мбайт/с (скорость работы считывающего устройства для компакт-дисков составляет 150 Кбайт/с). К сожалению, из-за перехода к красному лазеру DVD-проигрывателям требовался второй лазер для чтения существующих музыкальных и компьютерных компакт-дисков, что немного увеличивало их сложность и стоимость.

Достаточно ли 4,7 Гбайт? Может быть. Если использовать формат сжатия MPEG-2 (стандарт IS 13346), DVD-диск объемом 4,7 Гбайт может вместить полноэкранную видеозапись длительностью 133 минуты с высокой разрешающей способностью (720 × 480) вместе со звуком на 8 языках и субтитрами на 32 других языках. Около 92 % фильмов, снятых в Голливуде, по длительности меньше 133 минут. Тем не менее для некоторых приложений (например, мультимедийных игр или справочных изданий) может понадобиться больше места, к тому же Голливуд не прочь записывать по несколько фильмов на один диск. В результате появилось 4 формата DVD-дисков:

1. Односторонние однослойные диски (4,7 Гбайт).
2. Односторонние двухслойные диски (8,5 Гбайт).
3. Двухсторонние однослойные диски (9,4 Гбайт).
4. Двухсторонние двухслойные диски (17 Гбайт).

Зачем так много форматов? Если говорить кратко, основная причина — политика. Компании Philips и Sony считали, что нужно выпускать односторонние диски с двойным слоем, а Toshiba и Time Warner хотели производить двухсторонние диски с одним слоем. Philips и Sony думали, что покупатели не захотят переворачивать диски, а Time Warner полагала, что если поместить два слоя на одну сторону диска, он не будет работать. Компромиссное решение — удовлетворить все пожелания, а рынок уже сам определит, какой из вариантов выживет.

При двухслойной технологии на нижний отражающий слой помещается полуотражающий слой. В зависимости от того, где фокусируется лазер, он отражается либо от одного слоя, либо от другого. Чтобы обеспечить надежное считывание информации, лунки и площадки нижнего слоя делаются чуть большими по размеру, поэтому его емкость немного меньше, чем верхнего.

Двухсторонние диски создаются путем склеивания двух односторонних дисков толщиной по 0,6 мм каждый. Чтобы толщина всех версий была одинаковой, односторонний диск толщиной 0,6 мм приклеивается к пустой подложке (воз-

можно в будущем эта подложка будет содержать 133 минуты рекламы, в надежде, что покупатели заинтересуются, что на ней). Структура двухстороннего диска с двойным слоем показана на рис. 2.24.

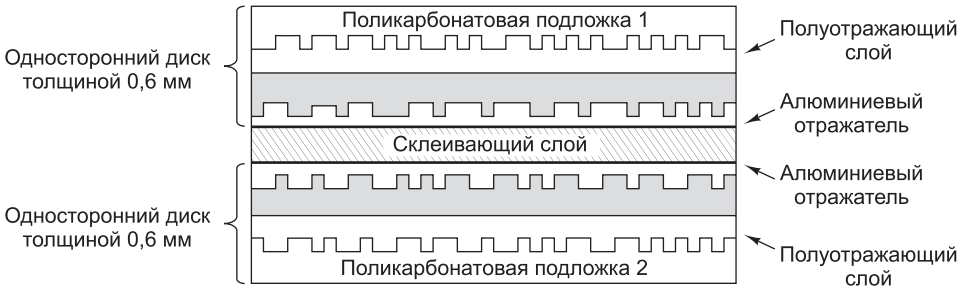


Рис. 2.24. Двухсторонний DVD-диск с двойным слоем

DVD-диск был разработан корпорацией, объединяющей 10 компаний по производству бытовой техники (семь из них японские), в тесном сотрудничестве с главными студиями Голливуда (владельцами некоторых из этих студий являлись японские компании). Ни компьютерная, ни телекоммуникационная промышленности не были вовлечены в разработку, в результате упор был сделан на использование DVD для проката и продажи фильмов. Перечислим некоторые стандартные особенности DVD: возможность исключать непристойные сцены из фильма (чтобы родители могли превратить фильм типа NC17[1] в фильм, который можно смотреть детям), шестиканальный звук, широкие возможности масштабирования. Последняя особенность позволяет DVD-проигрывателю решать, как обрезать правый и левый края рамки изображения у таких фильмов, у которых соотношение ширины и высоты составляет 3:2, чтобы их можно было без ущерба для качества воспроизводить на экранах современных телевизоров (с соотношение ширины и высоты 4:3).

Еще одна особенность, которая, вероятно, никогда не пришла бы в голову создателям компьютерных технологий, — намеренная несовместимость стандартов дисков для Соединенных Штатов, для европейских стран, для стран с других континентов. Голливуд ввел такую систему, потому что новые фильмы всегда сначала выпускаются на экраны в Соединенных Штатах и только после выпуска видеокассет отправляются в Европу. Это делается для того, чтобы европейские магазины видеопродукции не могли покупать видеозаписи в Америке слишком рано (что могло бы привести к снижению сборов новых фильмов в кинотеатрах Европы). Вероятно, если бы Голливуд стоял во главе компьютерной индустрии, в Америке дискеты были бы размером 3,5 дюйма, а в Европе — 9 см.

Диски Blu-Ray

Ничто не вечно в компьютерном бизнесе, особенно технологии хранения. Стоило только появиться стандарту DVD, как тут же у него обнаружился серьезный конкурент. Преемником DVD можно считать технологию **Blu-Ray**, предусма-

[1] NC17 — фильмы, содержащие сцены секса и насилия и не предназначенные для семейного просмотра. — *Примеч. перев.*

тривающую применение синего лазера вместо красного. Синий лазер отличается более короткой длиной волны, а значит, повышенной точностью; за счет этого обстоятельства он позволяет уменьшать размеры лунок и площадок. На односторонних дисках Blu-Ray умещается около 25 Гбайт данных; на двухсторонних — 50 Гбайт. Скорость передачи данных составляет 4,5 Мбит/с, что очень неплохо для оптических дисков, хотя по-прежнему несопоставимо с магнитными (напомним, стандарт ATAPI-6 предусматривает передачу данных на скорости 100 Мбит/с, а Ultra4 SCSI позволяет поднять скорость до 320 Мбит/с). Ожидается, что диски Blu-Ray в конечном счете вытеснят и CD-, и DVD-диски, но на это, конечно, уйдет не один год.

Ввод-вывод

Как отмечалось в начале этой главы, компьютерная система состоит из трех основных компонентов: центрального процессора, памяти (основной и вспомогательной) и **устройств ввода-вывода** (принтеров, сканеров и модемов). До сих пор мы рассматривали центральный процессор и память. Теперь мы поговорим об устройствах ввода-вывода и о том, как они соединяются с остальными компонентами системы.

Шины

Большинство персональных компьютеров и рабочих станций имеют физическую структуру, сходную с показанной на рис. 2.25. Обычно устройство представляет собой металлический корпус с большой интегральной схемой на дне, которая называется **материнской платой** (политкорректности ради можно называть ее системной платой). Материнская плата содержит микросхему процессора, несколько разъемов для модулей DIMM и различные вспомогательные микросхемы. Еще на материнской плате располагаются шина (она тянется вдоль платы) и несколько разъемов для подсоединения устройств ввода-вывода.

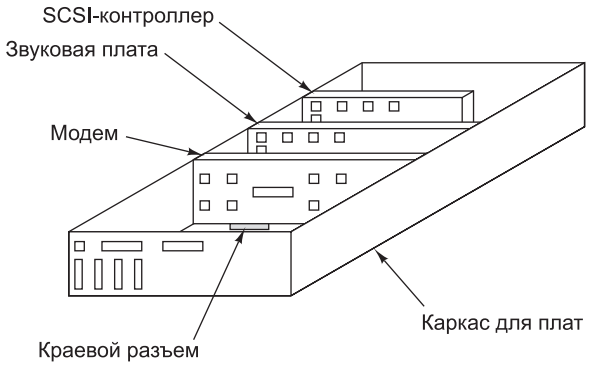


Рис. 2.25. Физическая структура персонального компьютера

Логическую структуру обычного персонального компьютера иллюстрирует рис. 2.26. У данного компьютера имеется одна шина для соединения централь-

ного процессора, памяти и устройств ввода-вывода; однако большинство систем имеют две и более шин. Каждое устройство ввода-вывода состоит из двух частей: одна объединяет большую часть электроники и называется **контроллером**, а другая представляет собой само устройство ввода-вывода, например дисковод. Контроллер обычно располагается на плате, которая вставляется в свободный разъем. Исключение представляют собой контроллеры устройств, являющихся неотъемлемыми составными частями компьютера (например, клавиатуры), которые иногда располагаются на материнской плате. Хотя дисплей (монитор) и нельзя назвать дополнительным устройством, соответствующий контроллер иногда располагается на встроенной плате, чтобы пользователь мог по желанию выбирать платы с графическими ускорителями или без них, устанавливать дополнительную память и т. д. Контроллер связывается с самим устройством кабелем, который соединяется с разъемом на задней стороне корпуса.

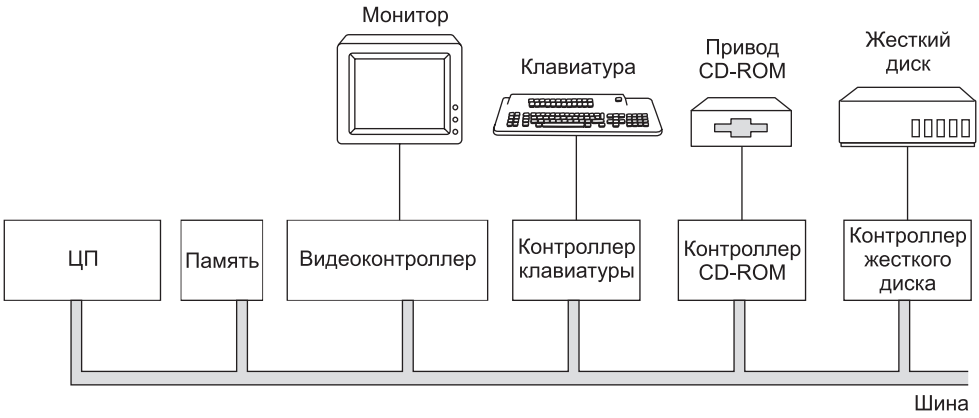


Рис. 2.26. Логическая структура обычного персонального компьютера

Контроллер управляет своим устройством ввода-вывода и для этого регулирует доступ к шине. Например, если программа запрашивает данные с диска, она посылает команду контроллеру диска, который затем отправляет диску команду поиска и другие команды. После нахождения соответствующей дорожки и сектора диск начинает передавать контроллеру данные в виде потока битов. Задача контроллера состоит в том, чтобы разбить поток битов на фрагменты и записывать каждый такой фрагмент по мере накопления битов для него в память. Отдельный фрагмент обычно представляет собой одно или несколько слов. Если контроллер считывает данные из памяти или записывает их в память без участия центрального процессора, то говорят, что осуществляется **прямой доступ к памяти** (Direct Memory Access, **DMA**). Когда передача данных заканчивается, контроллер выдает **прерывание**, вынуждая центральный процессор приостановить работу текущей программы и начать выполнение особой процедуры. Эта процедура называется **программой обработки прерываний** и нужна она для того, чтобы проверить, нет ли ошибок, в случае их обнаружения произвести необходимые действия и сообщить операционной системе, что процесс ввода-вывода завершен. Когда программа обработки прерывания завершается, процессор возобновляет работу программы, которая была приостановлена в момент прерывания.

Шина используется не только контроллерами ввода-вывода, но и процессором для передачи команд и данных. А что происходит, если процессор и контроллер ввода-вывода хотят получить доступ к шине одновременно? В этом случае особая микросхема, которая называется **арбитром шины**, решает, чья очередь первая. Обычно предпочтение отдается устройствам ввода-вывода, поскольку работу дисков и других движущихся устройств нельзя прерывать, так как это может привести к потере данных. Когда ни одного устройства ввода-вывода не функционирует, центральный процессор может полностью распоряжаться шиной для взаимодействием с памятью. Однако если работает какое-нибудь устройство ввода-вывода, оно будет запрашивать доступ к шине и получать его каждый раз, когда ему это необходимо. Этот процесс, который притормаживает работу компьютера, называется **захватом цикла памяти** (cycle stealing).

Описанная структура успешно использовалась в первых персональных компьютерах, поскольку все их компоненты работали примерно с одинаковой скоростью. Однако как только центральные процессоры, память и устройства ввода-вывода стали работать быстрее, возникла проблема: шина перестала справляться с нагрузкой. В случае закрытых систем, таких как инженерные рабочие станции, решением проблемы стала разработка для следующей модели машины новой шины с более высокой скоростью передачи данных. Поскольку в закрытых системах никто никогда не переносил устройства ввода-вывода со старой модели на новую, такой подход работал успешно.

Однако в мире персональных компьютеров большая часть пользователей, заменяя свой компьютер новой моделью, никак не рассчитывает одновременно отказываться от своих старых и привычных принтера, сканера и модема. Кроме того, существовала целая отрасль промышленности, выпускавшая широкий спектр устройств ввода-вывода для компьютеров IBM PC, и производители этих устройств совершенно не были заинтересованы в том, чтобы начинать все свои разработки заново. Компания IBM узнала об этом на горьком опыте, выпустив после линейки IBM PC линейку PS/2. У компьютеров PS/2 была новая шина с более высокой скоростью передачи данных, но большинство производителей клонов продолжали использовать старую шину PC, которая сейчас называется шиной **ISA** (Industry Standard Architecture — **стандартная промышленная архитектура**). Большинство производителей дисков и устройств ввода-вывода также продолжали выпускать контроллеры для старой модели, и компания IBM оказалась в весьма неприятной ситуации: на тот момент она оказалась единственным производителем персональных компьютеров, несовместимых с линейкой IBM. В конце концов компания была вынуждена вернуться к производству компьютеров на основе шины ISA. В наши дни шина ISA встречается разве что в самых древних системах и в музеях компьютерной техники, так как на смену ей пришли новые, более быстрые стандарты архитектуры шин. Отметим, что аббревиатура ISA также расшифровывается как Instruction Set Architecture (архитектура набора команд), если речь идет об уровнях иерархии команд.

Шины PCI и PCIe

Хотя влияние рынка было направлено на то, чтобы старая шина оставалась неизменной, быстрее она работать не стала, и нужно было что-то предпринять.

В результате другие компании начали производить компьютеры с несколькими шинами, одной из которых была либо прежняя шина ISA, либо шина **EISA** (Extended ISA — **расширенная стандартная промышленная архитектура**), как и ISA совместимая со старыми устройствами ввода-вывода. Что касается другой шины, то в настоящее время самой популярной моделью является шина **PCI** (Peripheral Component Interconnect — **взаимодействие периферийных компонентов**), разработанная компанией Intel, которая решила открыть всю связанную с шиной техническую информацию, чтобы сторонние производители (в том числе конкуренты компании) могли разрабатывать соответствующие устройства.

Существует много различных конфигураций шины PCI. Наиболее типичная из них показана на рис. 2.27. В такой конфигурации центральный процессор взаимодействует с контроллером памяти по выделенному высокоскоростному соединению. Таким образом, контроллер соединяется с памятью непосредственно, то есть передача данных между центральным процессором и памятью происходит не через шину PCI. Другие периферийные устройства подсоединяются прямо к шине PCI. Машина такого типа обычно содержит 2 или 3 пустых разъема PCI, чтобы покупатели имели возможность подключать карты PCI для новых периферийных устройств).

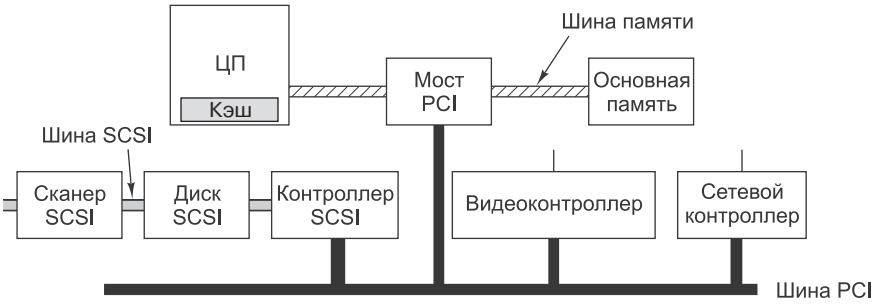


Рис. 2.27. Современный персональный компьютер с шиной PCI. Контроллер SCSI является PCI-устройством

Как бы быстро ни работало компьютерное оборудование, найдется много людей, которым оно покажется слишком медленным. Такая судьба постигла и шину PCI, которая была заменена шиной **PCI Express** (сокращенно **PCIe**). Многие современные компьютеры поддерживают обе шины, благодаря чему пользователи могут подключать новые, быстрые устройства к шине PCIe, а старые, более медленные — к шине PCI.

Если шина PCI представляла собой обновленную версию старой шины ISA с более высокой скоростью и разрядностью параллельно передаваемых данных, PCIe представляет кардинальное изменение по сравнению с шиной PCI. Собственно, это вообще не шина, а одноранговая сеть, использующая разрядно-последовательные линии и коммутацию пакетов. У нее больше от Интернета, чем от традиционных шин. Архитектура PCIe изображена на рис. 2.28.

Некоторые особенности шины PCIe сразу бросаются в глаза. Во-первых, соединения между устройствами являются последовательными, то есть имеют разрядность в один бит вместо 8, 16, 32 или 64 бит. Хотя казалось бы, 64-разряд-

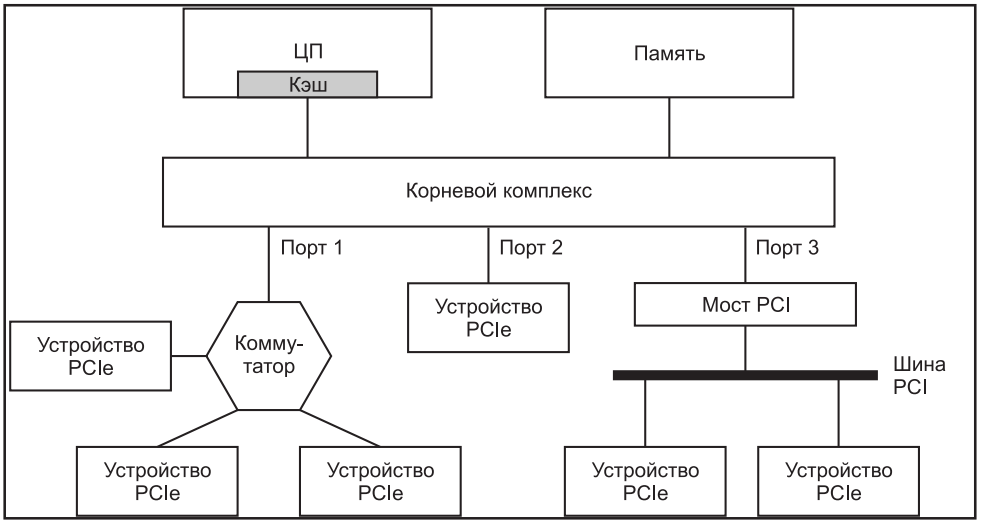


Рис. 2.28. Архитектура системы PCIe с тремя портами PCI

ное соединение обладает более высокой пропускной способностью, на практике различия во времени распространения 64-разрядной информации, называемые **расфазировкой**, заставляют использовать относительно низкие скорости передачи данных. По последовательному соединению данные передаются на значительно более высокой скорости, что более чем компенсирует потерю параллелизма. Шины PCI работают на максимальной тактовой частоте 66 МГц. При передаче 64 бит за такт скорость передачи данных составляет 528 Мбайт/с. При тактовой частоте 8 Гбит/с, даже в случае последовательной передачи, скорость передачи по шине PCIe составляет 1 Гбайт/с. Кроме того, обмен данными между устройством и корневым комплексом или коммутатором не ограничивается одной проводной парой. Устройство может иметь до 32 проводных пар, называемых **трактами** (lanes) или дорожками. Тракты работают несинхронно, поэтому расфазировка в данном случае несущественна. На большинстве материнских плат имеется 16-трактовый разъем для графической карты, что для PCIe 3.0 обеспечивает пропускную способность в 16 Гбайт/с — примерно в 30 раз больше, чем у графических карт PCI. Такая пропускная способность необходима для приложений, требования которых постоянно растут — например, трехмерной графики.

Во-вторых, все взаимодействия являются одноранговыми. Когда процессор хочет обратиться к устройству, он отправляет этому устройству пакет и обычно получает ответ. Пакет проходит через корневой комплекс на материнской плате, а затем передается устройству — как правило, через коммутатор (или для устройств PCI — через мост PCI). Переход от системы, в которой все устройства взаимодействуют с общей шиной, к системе с одноранговыми взаимодействиями, соответствует направлению развития Ethernet (популярная технология локальных сетей), которая в исходном варианте тоже использовала широковещательный канал, но в наши дни используется на одноранговых взаимодействиях с использованием коммутаторов.

Терминалы

В настоящее время существуют множество разнообразных устройств ввода-вывода. Мы коснемся только наиболее распространенных из них. Терминалы компьютера состоят из двух частей: клавиатуры и монитора. В мэйнфреймах эти части объединены в одно устройство и связаны с самим мэйнфреймом обычным или телефонным проводом. В авиакомпаниях, банках и различных отраслях промышленности, где работают мэйнфреймы, эти устройства до сих пор широко распространены. В мире персональных компьютеров клавиатура и монитор — независимые устройства, однако технологически клавиатура и монитор мэйнфрейма ничем не отличаются от соответствующих устройств ПК.

Клавиатуры

Существуют несколько видов клавиатур. У первых компьютеров IBM PC под каждой клавишей находился переключатель, который давал ощутимую отдачу и щелкал при нажатии клавиши. Сегодня механический контакт с печатной платой при нажатии клавиш происходит лишь у самых дешевых клавиатур. У клавиатур получше между клавишами и печатной платой располагается слой эластичного материала (особого типа резины). Под каждой клавишей находится небольшой купол, который прогибается в случае нажатия клавиши. Проводящий материал, находящийся внутри купола, замыкает схему. У некоторых клавиатур под каждой клавишей находится магнит, который при нажатии клавиши проходит через катушку и таким образом вызывает электрический ток. Используются и другие методы, как механические, так и электромагнитные.

В персональных компьютерах при нажатии клавиши происходит процедура прерывания и запускается программа обработки прерывания (эта программа является частью программного обеспечения операционной системы). Программа обработки прерывания считывает содержимое аппаратного регистра из контроллера клавиатуры, чтобы получить номер нажатой клавиши (от 1 до 102). Когда клавиша отпускается, происходит второе прерывание. Так, если пользователь нажимает клавишу SHIFT, затем нажимает и отпускает клавишу M, а после этого отпускает клавишу SHIFT, операционная система понимает, что ему нужна прописная, а не строчная буква M. Обработка нажатий клавиш SHIFT, CTRL и ALT в сочетании с другими клавишами выполняется только программно (сюда же относится известное сочетание клавиш CTRL+ALT+DEL, которое используется для перезагрузки всех компьютеров IBM PC и их клонов).

Сенсорные экраны

Хотя клавиатуры еще не собираются отправляться вслед за механическими пишущими машинками, в области компьютерного ввода появилась новая технология сенсорных экранов. Хотя эти устройства вышли на массовый рынок только с выходом Apple iPhone в 2007 году, появились они намного раньше. Первый сенсорный экран был разработан в фирме Royal Radar Establishment в Мэлверне, Великобритания, в 1965 году. Даже характерные жесты масштабирования сведением/разведением пальцев, так широко разрекламированные для iPhone, были изобретены в ходе работы, проводившейся в университете Торонто в 1982 году. С тех пор исследователи разработали и вывели на рынок много разных технологий.

Сенсорные устройства делятся на прозрачные и непрозрачные. Типичное непрозрачное сенсорное устройство — сенсорная панель (тачпад) на ноутбуке. Типичное прозрачное устройство — экран смартфона или планшетного компьютера. Мы ограничимся рассмотрением устройств второго типа, которые обычно называются **сенсорными экранами**. Основные разновидности сенсорных экранов — инфракрасные, резистивные и емкостные.

Принцип работы **инфракрасных** экранов основан на размещении инфракрасных передатчиков (скажем, инфракрасных светодиодов или лазеров) на левом и верхнем краях оправы, с детекторами на правом и нижнем краях. Когда палец, стилус или любой непрозрачный объект блокирует один или несколько лучей сетки, соответствующий детектор обнаруживает исчезновение сигнала. Оборудование устройства может сообщить операционной системе, какой из лучей был заблокирован; по этим данным вычисляются координаты (x, y) пальца или стилуса. Эта технология появилась уже давно, она до сих пор используется в интерактивных киосках и других областях, но в мобильных устройствах она не применяется.

Другая старая технология изготовления сенсорных экранов — **резистивная** — состоит из двух слоев. Верхний гибкий слой содержит большое количество горизонтальных проводников. В находящейся под ним мембране проходят вертикальные проводники. Когда палец или другой объект нажимает на экран, один из проводников верхней панели соприкасается (или подходит близко) к перпендикулярным проводникам нижней панели. Электроника устройства позволяет определить, в какой области было произведено нажатие. Резистивные экраны очень дешевы, они широко применяются в областях, критичных по цене.

Обе технологии хорошо работают при нажатии одним пальцем, но при использовании двух пальцев возникают проблемы. Для объяснения сути проблемы мы воспользуемся терминологией инфракрасного сенсорного экрана, но у резистивных экранов возникает та же проблема. Представьте, что два пальца нажимают на экран в точках (3, 3) и (8, 8). В результате прерываются вертикальные лучи $x = 3$ и $x = 8$, как и горизонтальные лучи $y = 3$ и $y = 8$.

Теперь рассмотрим другую ситуацию: пользователь нажимает на экран в точках (3, 8) и (8, 3) — противоположных углах прямоугольника с углами (3, 3), (8, 3), (8, 8) и (3, 8). При этом блокируются те же самые лучи, а программа не может определить, с какой из двух ситуаций она имеет дело. Эта проблема называется **двоением**.

Для обнаружения одновременных нажатий несколькими пальцами (свойство, необходимое для распознавания жестов сведения/разведения) потребовалась новая технология. В большинстве смартфонов и планшетных компьютерах (но не на цифровых камерах и других устройствах!) чаще всего используются **проекционно-емкостные** сенсорные экраны. Они тоже делятся на несколько разновидностей, наиболее распространенной из которых является **взаимно-емкостная**. Все сенсорные экраны, способные одновременно распознавать две и более точки контакта, называются **мультитач**-экранами. Давайте в общих чертах посмотрим, как они работают.

Для читателей, забывших школьный курс физики: конденсатор — устройство, способное накапливать электрический заряд. Простой конденсатор состоит из двух электродов в форме пластин, разделенных слоем диэлектрика. В совре-

менных сенсорных экранах сетка тонких «проводов», проходящих вертикально, отделяется от горизонтальной сетки тонким изолирующим слоем. Когда палец прикасается к экрану, он изменяет емкость всех затронутых пересечений (возможно, находящихся далеко друг от друга). Это изменение можно измерить. Чтобы убедиться в том, что современные сенсорные экраны отличаются от старых инфракрасных и резистивных экранов, попробуйте прикоснуться к экрану ручкой, карандашом, скрепкой или пальцем в перчатке — вы увидите, что ничего не происходит. Тело человека хорошо накапливает электрический заряд — каждый, кто в сухой холодный день вытирал ноги о коврик, а потом прикасался к металлической дверной ручке, знает это на собственном опыте. Пластмассовые, деревянные и металлические инструменты не могут сравниться с человеком в отношении своей емкости.

«Проводники» в сенсорном экране не похожи на обычные медные провода из обычных электрических устройств — они бы закрывали свет от экрана. Вместо них используются тонкие (обычно 50 микрон) полоски прозрачного резистивного сплава оксида индия и оксида олова, прикрепленные к обратным сторонам тонкой стеклянной панели. В совокупности они образуют конденсатор. В некоторых новых конструкциях диэлектрическая стеклянная панель заменяется тонким слоем диоксида кремния (песка!). В любом случае конденсаторы защищаются от грязи и царапин стеклянной пластиной, образующей поверхность экрана. Чем тоньше стеклянная пластина, тем чувствительнее экран, но и тем меньше прочность устройства.

В процессе работы устройства напряжение подается попеременно на горизонтальные и вертикальные проводники, в то время как с других проводников читаются значения напряжения, изменившиеся под воздействием емкости пересечения. Эта операция повторяется много раз за секунду, а координаты точки прикосновения передаются драйверу устройства в виде потока пар (x, y). Дальнейшая обработка (например, определение простого нажатия, жестов сведения/разведения или скольжения) выполняется операционной системой. Если вы используете все 10 пальцев, да еще позовете друга на помощь, операционной системе придется изрядно поломать голову, но мультитач-оборудование справится со своей задачей.

Плоские мониторы

В первых компьютерных мониторах использовались **электронно-лучевые трубки (ЭЛТ)**, как в старых телевизорах. Они были слишком громоздкими и тяжелыми для использования в портативных компьютерах, поэтому для экранов портативных компьютеров требовалась совершенно другая технология. Развитие плоских (flat-panel) мониторов позволило реализовать компактный форм-фактор, необходимый для ноутбуков, к тому же эти устройства использовали меньше энергии. В наши дни преимущества плоских экранов привели практически к полному вымиранию ЭЛТ-мониторов.

Самой распространенной технологией плоских мониторов является **жидкокристаллический дисплей**. Соответствующая технология чрезвычайно сложна, имеет несколько вариантов воплощения и быстро меняется, тем не менее мы постараемся сделать ее описание по возможности кратким и простым.

Жидкие кристаллы представляют собой вязкие органические молекулы, которые двигаются как молекулы жидкостей, но при этом имеют структуру, как у кристалла. Они были открыты австрийским ботаником Рейницером (Rheinitzer) в 1888 году и впервые стали применяться при изготовлении разнообразных дисплеев (для калькуляторов, часов и т. п.) в 1960 году. Когда молекулы расположены в одну линию, оптические качества кристалла зависят от направления и поляризации воздействующего света. При использовании электрического поля линия молекул, а следовательно, и оптические свойства, меняются. Если воздействовать лучом света на жидкий кристалл, интенсивность света, исходящего из самого жидкого кристалла, может контролироваться с помощью электричества. Это свойство используется при создании индикаторных дисплеев.

Экран жидкокристаллического дисплея состоит из двух стеклянных параллельно расположенных пластин, между которыми находится герметичное пространство с жидким кристаллом. К обеим пластинам подсоединяются прозрачные электроды. Искусственный или естественный свет за задней пластиной освещает экран изнутри. Электроды, подведенные к пластинам, используются для того, чтобы создать электрические поля в жидком кристалле. На различные части экрана воздействует разное напряжение, что и позволяет строить изображение. К передней и задней пластинам экрана приклеиваются поляроиды, поскольку технологически дисплей требует поляризованного света. Общая структура показана на рис. 2.29, *а*.

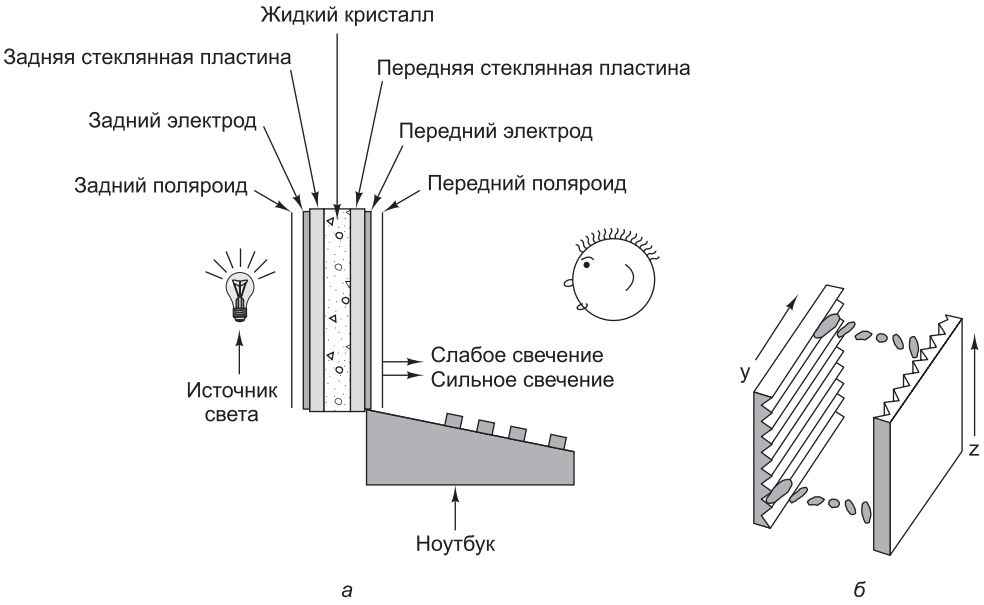


Рис. 2.29. Структура экрана на жидких кристаллах (*а*); желобки на передней и задней пластинах, расположенные перпендикулярно друг к другу (*б*)

В настоящее время используются различные типы жидкокристаллических дисплеев, но мы рассмотрим только один из них — дисплей со **скрученным нематиком** (Twisted Nematic, **TN**). В этом дисплее на задней пластине находятся

крошечные горизонтальные желобки, а на передней — крошечные вертикальные желобки, как показано на рис. 2.29, *б*. При отсутствии электрического поля молекулы направляются к этим желобкам. Так как они (желобки) расположены перпендикулярно друг к другу, молекулы жидкого кристалла оказываются скрученными на 90°.

На задней пластине дисплея находится горизонтальный поляроид. Он пропускает только горизонтально поляризованный свет. На передней пластине дисплея находится вертикальный поляроид. Он пропускает только вертикально поляризованный свет. Если бы между пластинами не было жидкого кристалла, горизонтально поляризованный свет, пропущенный поляроидом на задней пластине, блокировался бы поляроидом на передней пластине, что делало бы экран полностью черным.

Однако скрученная кристаллическая структура молекул, сквозь которую проходит свет, меняет плоскость поляризации света. При отсутствии электрического поля весь жидкокристаллический экран светится с однородной яркостью. Если подавать напряжение к определенным частям пластины, скрученная структура разрушается, блокируя прохождение света в этих частях.

Для подачи напряжения обычно используется два подхода. В дешевом **пассивном матричном индикаторе** на обоих электродах провода располагаются параллельно друг другу. Например, на дисплее размером 640 × 480 электрод задней пластины содержит 640 вертикальных проводов, а электрод передней пластины — 480 горизонтальных проводов. Если подавать напряжение на один из вертикальных проводов, а затем посылать импульсы на один из горизонтальных, можно изменить напряжение в определенной позиции пиксела и, таким образом, сделать нужную точку темной. Если то же самое повторить со следующим пикселом и т. д., можно получить темную строку развертки, аналогичную строкам в электронно-лучевых трубках. Обычно изображение на экране перерисовывается 60 раз в секунду, чтобы создавалось впечатление постоянной картинки (так же, как в электронно-лучевых трубках).

Второй подход — применение **активного матричного индикатора**. Он стоит гораздо дороже, чем пассивный, но зато дает изображение лучшего качества, что является большим преимуществом. Вместо двух наборов перпендикулярно расположенных проводов у активного матричного индикатора на одном из электродов имеется крошечный переключатель в каждой позиции пиксела. Меняя состояние переключателей, можно создавать на экране произвольную комбинацию напряжений в зависимости от комбинации битов. Эти переключатели называются **тонкопленочными транзисторами** (Thin Film Transistor, **TFT**), а плоские экраны, в которых они используются, — **TFT-дисплеями**. На основе технологии TFT теперь производится подавляющее большинство ноутбуков и автономных жидкокристаллических мониторов.

До сих пор мы описывали, как работают монохромные мониторы. Что касается цветных мониторов, достаточно сказать, что они работают на основе тех же общих принципов, что и монохромные, но детали гораздо сложнее. Чтобы разделить белый цвет на красный, зеленый и синий, в каждой позиции пиксела используются оптические фильтры, поэтому эти цвета могут отображаться независимо друг от друга. Из сочетания этих трех основных цветов можно получить любой цвет.

В ближайшем будущем появятся и другие экранные технологии. Весьма перспективна технология органических светодиодов **OLED** (Organic Light Emitting Diode). Такие экраны состоят из слоев электрически заряженных органических молекул, помещенных между двумя электродами. Изменения напряжения заставляют молекулы переходить на более высокие энергетические состояния. При возвращении к нормальному состоянию они излучают свет. Более подробное описание выходит за рамки книги (и познаний авторов).

Видеопамять

Обновление картинки на экранах ЭЛТ- и TFT-мониторов производится от 60 до 100 раз в секунду; для этого используется видеопамять, размещенная на плате контроллера дисплея. Видеопамять содержит одну или несколько битовых карт, представляющих выводимое на экран изображение. Если, скажем, на экране умещается 1920 × 1080 элементов изображения **(пикселов)**, значит, в видеопамяти содержится 1920 × 1080 значений, по одному на каждый пиксел. В целях быстрого переключения с одного изображения на другое в памяти может размещаться несколько таких карт.

В современных дисплеях каждый пиксел представлен 3-байтным значением **RGB**, которое определяет интенсивность красного (Red), зеленого (Green) и синего (Blue) компонентов изображения (мощные профессиональные мониторы используют 10 и более бит на цвет). Как известно, любой цвет можно представить путем линейной суперпозиции трех упомянутых базовых цветов.

Если в видеопамяти хранится информация о 1920 × 1080 пикселах, причем на каждый из них выделяется по 3 байта, общий объем этих данных составляет около 6,2 Мбайт; поэтому на любые манипуляции таким изображением уходит довольно много процессорного времени. По этой причине в некоторых компьютерах для определения цвета используются 8-разрядные числа. Такое число представляет собой индекс аппаратной таблицы (так называемой **цветовой палитры**), состоящей из 256 значений RGB (24-разрядных). Это решение, известное под названием **индексированного цвета**, позволяет на 2/3 сократить объем данных, хранящихся в видеопамяти. В то же время, при применении индексированного цвета в каждый конкретный момент на экран не может выводиться более 256 цветов. Как правило, для каждого окна формируется индивидуальная битовая карта, а это значит, что при наличии одной аппаратной палитры из всех присутствующих на экране окон корректно визуализируется только одно. Также применяются палитры с 2^{16} элементами, но в этом случае выигрыш по занимаемой памяти составляет всего 1/3.

Для вывода растровых (то есть сформированных на основе битовых карт) изображений требуется большая пропускная способность. К примеру, для воспроизведения одного кадра полноцветных мультимедийных данных в полноэкранном формате на дисплее размером 1920 × 1080 необходимо скопировать в видеопамять 6,2 Мбайт. Если учесть, что полноценный видеофильм выводится со скоростью 25 кадров в секунду, общая скорость передачи данных должна составлять 155 Мбайт/с. Такую пропускную способность не способна обеспечить даже первоначальная версия шины PCI (132 Мбайт/с), но шина PCIe легко справляется с ней.

Мыши

Время идет, и за компьютер садятся те, кто разбирается в нем все меньше и меньше. Компьютеры серии ENIAC использовались только теми, кто их разрабатывал. В 50-е годы с компьютерами работали лишь высоко квалифицированные программисты. Сейчас многие из тех, кто работает за компьютером, не знают (и не хотят знать) ни как функционирует компьютер, ни как он программируется.

Много лет назад у большинства компьютеров был интерфейс командной строки, в которой набирались различные команды. Поскольку многие неспециалисты считали такие интерфейсы недружественными или даже враждебными, компьютерные фирмы разработали специальные интерфейсы с возможностью указания определенной позиции на экране с помощью специального устройства (как в Macintosh и Windows), которым чаще всего является мышь.

Мышь — это устройство в маленьком пластиковом корпусе, располагающееся на столе рядом с клавиатурой. Если двигать мышь по столу, указатель на экране тоже будет двигаться, что дает возможность навести его на тот или иной элемент экрана. У мыши есть одна, две или три кнопки, нажатие которых дает возможность пользователям выбирать пункты меню. Было очень много споров по поводу того, сколько кнопок должно быть у мыши. Начинающим пользователям достаточно было одной кнопки (в этом случае перепутать кнопки невозможно), но их более опытные коллеги предпочитали несколько кнопок, чтобы можно было на экране выполнять сложные действия.

Существует три типа мышей: механические, оптические и оптомеханические. У мышей первого типа снизу располагаются резиновые колесики, оси которых расположены перпендикулярно друг к другу. Если мышь передвигается в вертикальном направлении, то вращается одно колесо, а если в горизонтальном, то другое. Каждое колесико приводит в действие резистор (потенциометр). Если измерить изменения сопротивления, можно узнать, на сколько провернулось колесико, и таким образом вычислить, на какое расстояние передвинулась мышь в каждом направлении. В последние годы такие мыши практически полностью вытеснены новой моделью, в которой вместо колес используется шарик, слегка выступающий снизу (рис. 2.30).

Следующий тип — оптическая мышь. У нее нет ни колес, ни шарика. Вместо этого в нижней части мыши располагаются **светодиод** и фотодетектор. Первые модели оптическах мышей перемещались по поверхности особого пластикового коврика, который содержит прямоугольную решетку с линиями, близко расположенными друг к другу. Когда мышь двигается по решетке, фотодетектор воспринимает пересечения линий за счет изменения в количестве света, отражаемого от светодиода. Электронное устройство внутри мыши подсчитывает количество пересеченных линий в каждом направлении. Современные оптические мыши содержат светодиод, освещающий неоднородности нижележащей поверхности, и крошечную видеокамеру, которая снимает изображение (как правило, размером 18×18 пикселов) до 1000 раз в секунду. Сравнение соседних изображений определяет, как далеко переместилась мышь. Некоторые оптические мыши используют для освещения лазер вместо светодиода. Они обеспечивают большую точность, но и стоят дороже.

Третий тип — оптомеханическая мышь. У нее, как и у более современной механической мыши, есть шарик, который вращает два колесика, расположенные

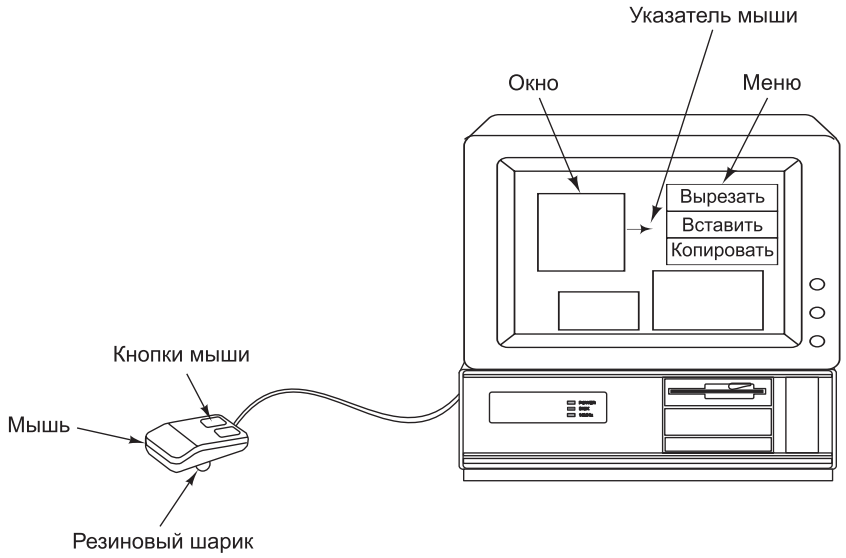


Рис. 2.30. Использование мыши для выбора пункта меню

перпендикулярно друг к другу. Колесики связаны с кодировщиками. В каждом кодировщике имеются прорези, через которые проходит свет. Когда мышь двигается, колесики вращаются, и световые импульсы воздействуют на детекторы каждый раз, когда между светодиодом и детектором появляется прорезь. Число воспринятых детектором импульсов пропорционально расстоянию.

Хотя мыши могут быть устроены по-разному, обычно используется следующая схема: компьютеру передается последовательность из 3 байт каждый раз, когда мышь проходит определенное минимальное расстояние (например, 0,01 дюйма). Обычно эти характеристики передаются в последовательном потоке битов. Первый байт содержит целое число, которое указывает, на какое расстояние переместилась мышь в направлении x с прошлого раза. Второй байт содержит ту же информацию для направления y. Третий байт указывает на текущее состояние кнопок мыши. Иногда для каждой координаты используются 2 байта.

Низкоуровневое программное обеспечение принимает эту информацию по мере поступления и преобразует относительные движения, передаваемые мышью, в абсолютную позицию. Затем оно отображает стрелочку на экране в позиции, соответствующей расположению мыши. Если указать стрелочкой на определенный элемент экрана и щелкнуть кнопкой мыши, компьютер может вычислить, какой именно элемент на экране выбран.

Игровые контроллеры

Видеоигры обычно предъявляют особенно высокие требования к взаимодействию с пользователем. Для рынка игровых приставок были разработаны специализированные устройства ввода. В этом разделе мы рассмотрим две интересные новинки в области контроллеров для видеоигр: Nintendo Wiimote и Microsoft Kinect.

Wiimote

Контроллер Wiimote, выпущенный в 2006 году для игровой приставки Nintendo Wii, содержит традиционные кнопки вместе с двойным датчиком перемещения. Все действия с Wiimote передаются в реальном времени игровой приставке через внутренний передатчик Bluetooth. Датчики перемещения позволяют Wiimote отслеживать перемещения в трех измерениях, а также обеспечивают точное распознавание направления при наведении на телевизор.

На рис. 2.31 показано, как Wiimote реализует функцию получения информации о параметрах движения. Отслеживание перемещений Wiimote в трехмерном пространстве обеспечивается внутренним 3-осевым акселерометром. Устройство содержит три небольших массы, каждая из которых может перемещаться по осям *x*, *y* и *z* (относительно микросхемы акселерометра). Движение масс осуществляется пропорционально ускорению по соответствующей оси, что приводит к изменению емкости массы по отношению к металлической стене. Измерение трех изменяющихся емкостей позволяет вычислить ускорения по трем направ-

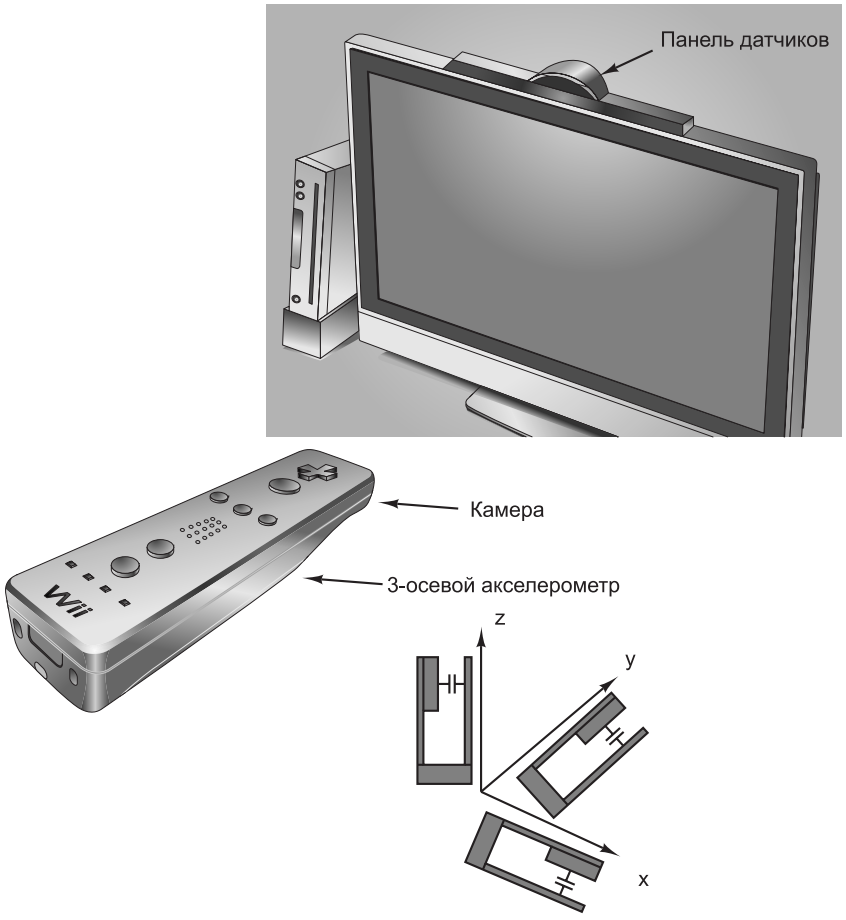


Рис. 2.31. Датчики движения игрового контроллера Wiimote

лениям. При помощи этой технологии и некоторых классических формул приставка Wii может отслеживать пространственные перемещения Wiimote. Когда игрок взмахивает Wiimote, чтобы ударить по виртуальному теннисному мячу, движение отслеживается в процессе взмаха. А если игрок в последний момент повернет кисть, чтобы придать мячу верхнее вращение, акселерометры Wiimote воспримут и это движение.

Хотя акселерометры хорошо справляются с отслеживанием Wiimote в трех направлениях, они не обеспечивают точности, необходимой для управления указателем на экране телевизора. Акселерометры страдают от неизбежных микроошибок при измерении ускорения, соответственно погрешность точного местонахождения Wiimote (основанного на объединении данных ускорения) неуклонно возрастает.

Для высокоточного распознавания движения Wiimote использует специальную технологию «компьютерного зрения». На телевизоре размещается панель датчиков (Sensor Bar) со светодиодами, разнесенными на фиксированное расстояние. В Wiimote находится миниатюрная камера, которая при наведении на панель датчиков может вычислить расстояние и ориентацию устройства по отношению к телевизору. Так как расстояние между светодиодами панели датчиков фиксировано, расстояние между ними в восприятии Wiimote пропорционально расстоянию Wiimote от панели. Расположение панели датчиков в поле зрения Wiimote определяет направление, в котором указывает Wiimote относительно телевизора. Непрерывное отслеживание ориентации обеспечивает высокую точность определения направления без позиционных ошибок, присущих акселерометрам.

Kinect

Microsoft Kinect поднимает возможности «компьютерного зрения» игровых контроллеров на совершенно новый уровень. Устройство использует для определения взаимодействий пользователя с игровой приставкой только технологию распознавания образов, и ничего более. Его работа основана на определении позиции пользователя в комнате, ориентации и движения его тела. Пользователь совершает заранее определенные движения руками, кистями и вообще всем, чем по мнению разработчиков он должен размахивать, чтобы управлять их игрой.

Функциональность Kinect обеспечивается датчиком глубины в сочетании с видеокамерой. Датчик глубины вычисляет расстояние до объекта, находящегося в поле зрения Kinect. Для этого он излучает матрицу инфракрасных лазерных точек, а затем читает их отражения на инфракрасную камеру. Используя технологию распознавания образов, которая называется «структурированным освещением», Kinect может определить расстояние до объектов в своем поле зрения по искажению матрицы инфракрасных точек освещенными поверхностями.

Данные глубины объединяются с информацией текстур, полученной с видеокамеры. В результате формируется текстурная карта глубины. Она обрабатывается алгоритмами распознавания образов для обнаружения людей, находящихся в комнате (даже с распознаванием их лиц), ориентации и движений их тел. После обработки информация о находящихся в комнате людях передается игровой приставке, которая использует ее для управления видеоигрой.

Принтеры

Рано или поздно пользователю потребуется напечатать созданный документ или страницу, полученную из Интернета, поэтому компьютеры могут быть оснащены принтером. В этом разделе мы опишем некоторые наиболее распространенные типы монохромных (то есть черно-белых) и цветных принтеров.

Лазерные принтеры

Вероятно, самым удивительным изобретением в области печатных технологий со времен Йоганна Гуттенберга (Johann Gutenberg), который изобрел подвижную литеру в XV веке, является **лазерный принтер**. Это устройство сочетает хорошее качество печати, универсальность, высокую скорость работы и умеренную стоимость. В лазерных принтерах используется почти та же технология, что и в фотокопировальных устройствах. Многие компании производят устройства, совмещающие свойства копировальной машины, принтера и иногда факса.

Схематически устройство принтера показано на рис. 2.32. Главной частью этого принтера является вращающийся барабан (в некоторых более дорогостоящих системах вместо барабана используется лента). Перед печатью каждого листа барабан получает напряжение около 1000 вольт и окружается фоточувствительным материалом. Свет лазера проходит вдоль барабана (по длине), почти как пучок электронов в электронно-лучевой трубке, только вместо напряжения для сканирования барабана используется вращающееся восьмиугольное зеркало. Луч света модулируется, в результате получается набор темных и светлых участков. Участки, на которые воздействует луч, теряют свой электрический заряд.

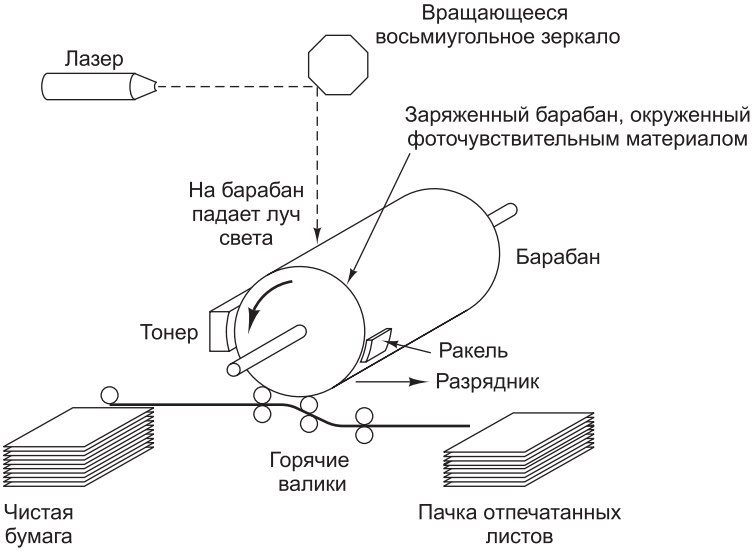


Рис. 2.32. Схема работы лазерного принтера

После того как нарисована строка точек, барабан немного поворачивается для создания следующей строки. В итоге первая строка точек достигает резервуара с тонером (электростатическим черным порошком). Тонер притягивается к заряженным точкам, и так формируется визуальное изображение строки. Через

некоторое время барабан с тонером прижимается к бумаге, оставляя на ней отпечаток изображения. Затем лист проходит через нагретые валики, и изображение закрепляется. После этого барабан разряжается и остатки тонера счищаются с него. Теперь он готов к печати следующей страницы.

Едва ли нужно говорить, что этот процесс представляет собой чрезвычайно сложную комбинацию приемов, требующих знания физики, химии, механики и оптики. Впрочем, некоторые фирмы предлагают готовые модули, называемые **блоками печати** (print engines). Производители лазерных принтеров дополняют блоки печати собственной электроникой и программным обеспечением. Электроника лазерных принтеров состоит из быстродействующего процессора и нескольких мегабайтов памяти для хранения полного изображения в битовой форме и различных шрифтов, одни из которых встроены, а другие загружаются из памяти. Большинство принтеров получают команды, описывающие печатаемую страницу (в противоположность принтерам, получающим изображения в битовой форме от центрального процессора). Эти команды обычно даются на языке PCL от HP или PostScript от Adobe — полноценных, хотя и специализированных, языках программирования.

Лазерные принтеры с разрешающей способностью 600 dpi и выше могу печатать черно-белые фотографии, но технология при этом гораздо сложнее, чем может показаться на первый взгляд. Рассмотрим фотографию, отсканированную с разрешающей способностью 600 dpi, которую нужно напечатать на принтере с такой же разрешающей способностью (600 dpi). Сканированное изображение содержит 600 × 600 пикселов на дюйм, каждый пиксел характеризуется определенной градацией серого цвета от 0 (белый цвет) до 255 (черный цвет). Принтер может печатать с разрешающей способностью 600 dpi, но каждый напечатанный пиксел может быть либо черного цвета (когда есть тонер), либо белого цвета (когда нет тонера). Градации серого печататься не могут.

При печати таких изображений имеет место так называемая **обработка полутонов** (как при печати серийных плакатов). Изображение разбивается на ячейки, каждая по 6 × 6 пикселов. Каждая ячейка может содержать от 0 до 36 черных пикселов. Человеческому глазу ячейка с большим количеством черных пикселов кажется темнее, чем ячейка с небольшим количеством черных пикселов. Серые тона в диапазоне от 0 до 255 передаются следующим образом. Этот диапазон делится на 37 зон. Серые тона от 0 до 6 расположены в зоне 0, от 7 до 13 — в зоне 1 и т. д. (зона 36 немного меньше, чем другие, потому что 256 на 37 без остатка не делится). Когда встречаются тона зоны 0, ячейка оставляется белой, как показано на рис. 2.33, *а*. Тона зоны 1 передаются одним черным пикселом в ячейке. Тона зоны 2 — двумя пикселами в ячейке, как показано на рис. 2.33, *б*. Изображения серых тонов других зон показаны на рис. 2.33, *в*–*е*. Если фотография отсканиро-

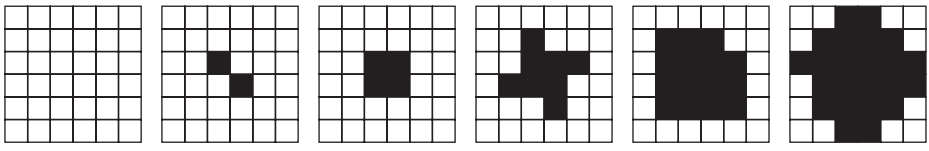


Рис. 2.33. Изображение серых полутонов различных зон: 0–6 (*а*); 14–20 (*б*); 28–34 (*в*); 56–62 (*г*); 105–111 (*д*); 161–167 (*е*)

вана с разрешающей способностью 600 dpi, после подобной обработки полутонов фактическая разрешающая способность напечатанного изображения снижается до 100 ячеек на дюйм. Данная величина называется **градацией полутонов** и измеряется в **lpi** (lines per inch — строк на дюйм).

Цветные принтеры

Хотя лазерные принтеры чаще всего являются монохромными, цветные лазерные принтеры получают все более широкое распространение, поэтому о цветной печати тоже стоит сказать пару слов (причем все сказанное также относится к струйным и другим принтерам). Цветные изображения могут строиться двумя способами: испусканием света и отражением света. Испускание света имеет место, например, при создании изображений в электронно-лучевых мониторах. В данном случае изображение строится путем аддитивного наложения трех основных цветов: красного, зеленого и синего.

Отраженный свет используется при создании цветных фотографий и картинок в глянцевых журналах. В этом случае поглощается свет с определенной длиной волны, а остальной свет отражается. Такие изображения создаются путем субтрактивного наложения трех основных цветов: голубого (красный полностью поглощен), желтого (синий полностью поглощен) и сиреневого (зеленый полностью поглощен). Теоретически путем смешения голубых, желтых и сиреневых чернил можно получить любой цвет. Но на практике очень сложно получить такие чернила, которые полностью поглощали бы весь свет и в результате давали черный цвет. По этой причине практически во всех цветных печатающих устройствах используются чернила четырех цветов: голубого (Cyan), сиреневого (Magenta), желтого (Yellow) и черного (blacK). Такая цветовая модель называется **CMYK** (из слова «black» берется последняя буква, чтобы отличать его от слова «blue» в модели RGB). Мониторы, напротив, для создания цветного изображения используют испускаемый свет и наложение красного, зеленого и синего цветов.

Полный набор цветов, который может производить монитор или принтер, называется **цветовой шкалой**. Не существует такого устройства, которое полностью передавало бы цвета окружающего нас мира. В лучшем случае устройство дает всего 256 степеней интенсивности каждого цвета, и в итоге получается только 16 777 216 различных цветов. Несовершенство технологий еще больше сокращает это число, а оставшиеся цвета не дают полного цветового спектра. Кроме того, цветовосприятие связано не только с физическими свойствами света, но и с работой «палочек» и «колбочек» в сетчатке глаза.

Из всего этого следует, что превратить красивое цветное изображение, которое замечательно смотрится на экране, в идентичное печатное изображение очень сложно. Среди основных проблем можно назвать следующие:

- цветные мониторы используют поглощенный свет; цветные принтеры — отраженный;
- электронно-лучевая трубка дает 256 оттенков каждого цвета, цветные принтеры должны обеспечивать обработку полутонов;
- мониторы имеют темный фон; бумага — светлый;
- цветовая модель RGB монитора и модель CMYK принтера отличаются друг от друга.

Чтобы цветные печатные изображения соответствовали реальной действительности (или хотя бы изображениям на экране), необходима калибровка оборудования, сложное программное обеспечение и компетентность пользователя.

Струйные принтеры

Дома удобно использовать недорогие **струйные принтеры**. В таком принтере подвижная печатающая головка содержит картридж с чернилами. Она двигается горизонтально над бумагой, а чернила в это время выпрыскиваются из крошечных сопел. Объем одной порции чернил приблизительно равен один пиколитр. Для наглядности уточним, что в одной капле воды может уместится около 100 миллионов таких порций.

Струйные принтеры бывают двух типов: пьезоэлектрические (производятся Epson) и термографические (производятся Canon, HP и Lexmark). В пьезоэлектрических струйных принтерах рядом с чернильной камерой устанавливается специальный кристалл. При подаче на этот кристалл напряжения он деформируется, в результате из форсунки выпускаются чернила. Чем выше напряжение, тем больше выходная порция чернил, причем управление этим процессом производится программно.

В термографических (пузырьковых) струйных принтерах в каждой форсунке устанавливается небольшой резистор. При подаче напряжения резистор быстро нагревается, доводит температуру чернил до точки кипения, в результате последние превращаются в пузырьки газа. Поскольку объем пузырька больше объема чистых чернил, в форсунке создается повышенное давление, под влиянием которого чернила распыляются на бумагу. Затем форсунка охлаждается, и в результате снижения давления внутри форсунки в нее из картриджа подается новая порция чернил. Скорость работы принтера в рамках этой схемы ограничена временными рамками цикла кипения/охлаждения. Размер всех формируемых чернильных капель одинаков, причем, как правило, он уступает аналогичному показателю пьезоэлектрических принтеров.

Струйные принтеры обычно имеют разрешающую способность от 1200 dpi (dots per inch — точек на дюйм) до 4800 dpi. Они достаточно дешевы, работают бесшумно, однако отличаются низкой скоростью печати и дороговизной картриджей. Качество печати хорошее — если распечатать фотографию с высоким разрешением на ведущей модели любой линейки струйных принтеров, результат будет не отличить от обычной фотографии формата 8 × 10.

Для получения лучших результатов должны использоваться особые чернила и особая бумага. Существует два вида чернил. **Чернила на основе красителя** состоят из красителей, растворенных в жидкой среде. Они дают яркие цвета и легко вытекают из картриджа. Главным недостатком таких чернил является то, что они быстро выгорают под воздействием ультрафиолетовых лучей, которые содержатся в солнечном свете. **Чернила на основе пигмента** содержат твердые частицы пигмента, погруженные в жидкость. Жидкость испаряется с бумаги, а пигмент остается. Чернила не выгорают, но зато дают не такие яркие краски, как чернила на основе красителя. Кроме того, частицы пигмента часто засоряют выпускные отверстия картриджей, поэтому их нужно периодически чистить. Для печати фотографий необходима мелованная или глянцевая бумага. Эти особые виды бумаги созданы специально для того, чтобы удерживать капельки чернил и не давать им растекаться.

Специализированные принтеры

Лазерные и струйные принтеры лидируют на рынке домашней и офисной печати. Однако существуют и другие технологии печати, применяемые в других ситуациях с другими требованиями по качеству цвета, цене и другим характеристикам.

Следующий тип принтеров — **принтеры с твердыми чернилами**. В этих принтерах содержится 4 твердых блока специальных восковых чернил, которые затем расплавляются, для чего перед началом печати должно пройти 10 минут (время, необходимое для того, чтобы расплавить чернила). Горячие чернила выпрыскиваются на бумагу, где они затвердевают и закрепляются после прохождения листа между двумя валиками. В каком-то смысле эта технология объединяет принципы работы струйных принтеров (напыление чернил) и лазерных принтеров (закрепление изображения на бумаге при помощи твердых резиновых роликов).

Принтеры с восковыми чернилами содержат широкую ленту из четырехцветного воска, которая разделяется на отрезки размером с лист бумаги. Тысячи нагревательных элементов растапливают воск, когда бумага проходит под лентой. Воск закрепляется на бумаге в форме пикселов в соответствие с цветовой моделью CMYK. Такие принтеры когда-то были очень популярными, но сейчас их вытеснили другие типы принтеров с более дешевыми расходными материалами.

Еще одна разновидность принтеров работает на основе технологии **сублимации**. Название вызывает фрейдистские ассоциации[1], однако в науке под сублимацией понимается переход твердых веществ в газообразное состояние минуя стадию жидкости. Таким материалом является, например, сухой лед (замороженный углекислый газ). В принтере, работающем на основе процесса сублимации, контейнер с красителями CMYK двигается над термической печатающей головкой, которая содержит тысячи программируемых нагревательных элементов. Красители мгновенно испаряются и впитываются специальной бумагой. Каждый нагревательный элемент может производить 256 различных температур. Чем выше температура, тем больше красителя осаждается и тем интенсивнее получается цвет. В отличие от всех других цветных принтеров, данный принтер способен воспроизводить цвета практически сплошного спектра, поэтому процедура обработки полутонов не нужна. Процесс сублимации часто используется при изготовлении так называемых моментальных снимков на специальной дорогостоящей бумаге.

Последнюю разновидность составляют **термографические принтеры**. Они содержат небольшую печатающую головку с множеством игольчатых элементов. При прохождении электрического тока иглы очень быстро нагреваются. Над печатающей головкой проходит специальная термочувствительная бумага, и в тех местах, где находятся нагретые иглы, появляются точки. В сущности, термографический принтер работает по принципу старого матричного принтера, в котором контакты через красящую ленту оставляли точки на бумаге. Термографические принтеры широко применяются для печати чеков в магазинах, банкоматах, автоматизированных заправках и т. д.

---

[1] Сублимация в психологии означает психический процесс преобразования и переключения энергии влечений на цели социальной деятельности и культурного творчества; термин введен З. Фрейдом. — *Примеч. перев.*

Телекоммуникационное оборудование

Большинство современных компьютеров подключаются к компьютерным сетям, из которых наиболее распространен Интернет. Для доступа к подобного рода сетям требуется специальное оборудование. В этом разделе рассматриваются принципы работы такого оборудования.

Модемы

С ростом количества компьютеров в последние годы возникла необходимость связать их между собой. Например, можно связать свой домашний компьютер с компьютером на работе, с поставщиком услуг Интернета или банковской системой. Для обеспечения такой связи часто используется телефонная линия.

Однако обычная телефонная линия (равно как и кабель) плохо подходит для передачи компьютерных сигналов, в которых 0 обычно соответствует нулевому напряжению, а 1 — напряжению от 3 до 5 вольт (рис. 2.34, *а*). Двухуровневые сигналы во время передачи по телефонной линии, которая предназначена для передачи голоса, подвергаются сильным искажениям, ведущим к ошибкам в передаче. Тем не менее синусоидальный сигнал с частотой от 1000 до 2000 Гц, который называется **несущим**, может передаваться с относительно небольшими искажениями, и это свойство используется при передаче данных в большинстве телекоммуникационных систем.

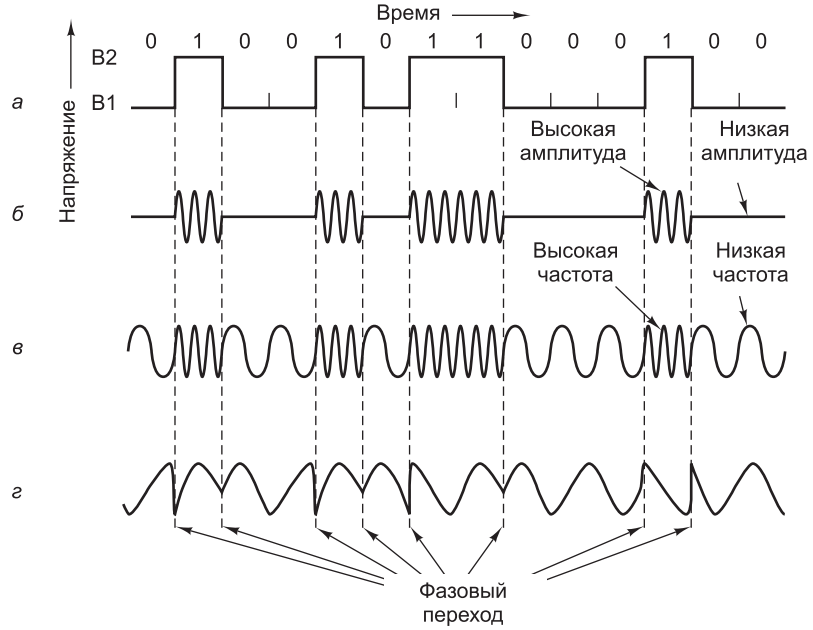


Рис. 2.34. Последовательная передача двоичного числа 01001011000100 по телефонной линии: двухуровневый сигнал (*а*); амплитудная модуляция (*б*); частотная модуляция (*в*); фазовая модуляция (*г*)

Поскольку форма синусоидальной волны полностью предсказуема, она не несет никакой информации. Однако изменяя амплитуду, частоту или фазу, можно

передавать последовательность нулей и единиц, как показано на рис. 2.34. Этот процесс называется **модуляцией**. При **амплитудной модуляции** используются 2 уровня напряжения, соответственно для 0 и 1 (рис. 2.34, *б*). Если цифровые данные передаются с очень низкой скоростью, то при передаче 1 слышен громкий шум, а при передаче 0 шум отсутствует.

При **частотной модуляции** уровень напряжения не меняется, но частоты несущего сигнала для 1 и 0 различаются (рис. 2.34, *в*). В этом случае при передаче цифровых данных можно услышать два тона: один из них соответствует 0, а другой — 1. Частотную модуляцию иногда называют **частотной манипуляцией**.

При простой **фазовой модуляции** амплитуда и частота сохраняются на одном уровне, а фаза несущего сигнала меняется на 180°, когда данные меняются с 0 на 1 или с 1 на 0 (рис. 2.34, *г*). В более сложных системах фазовой модуляции в начале каждого неделимого временного отрезка фаза несущего сигнала резко сдвигается на 45, 135, 225 или 315°, чтобы передавать 2 бита за один временной отрезок. Это называется **дибитной фазовой кодировкой**. Например, сдвиг по фазе на 45° представляет 00, на 135° — 01 и т. д. Существуют системы для передачи трех и более битов за один временной отрезок. Число таких временных интервалов (то есть число потенциальных изменений сигнала в секунду) называется скоростью в **бодах**. При передаче двух или более битов за один временной отрезок скорость передачи битов будет превышать скорость в **бодах**. Отметим, что термины «бод» и «бит» в этом контексте часто путают. Еще раз: скорость в бодах определятся количеством изменений сигнала за секунду, а скорость в битах — количеством битов, переданных за секунду. Как правило, скорость в битах кратна скорости в бодах, но теоретически может быть и ниже.

Если данные состоят из последовательности 8-разрядных символов, было бы желательно иметь средство связи для передачи 8 бит одновременно, то есть 8 пар проводов. Так как телефонные линии, предназначенные для передачи голоса, обеспечивают только один канал связи, биты должны пересылаться последовательно один за другим (или в группах по два, если используется дибитная кодировка). Устройство, которое получает символы из компьютера в форме двухуровневых сигналов (по одному биту в каждый отрезок времени) и передает биты по одному или по два в форме амплитудной, фазовой или частотной модуляции, называется модемом. Для указания на начало и конец каждого символа в начале и конце 8-разрядной цепочки ставятся начальный и конечный биты, таким образом, всего получается 10 бит.

Модем посылает отдельные биты каждого символа через равные временные отрезки. Например, скорость 9600 бод означает, что сигнал меняется каждые 104 микросекунд. Второй модем, получающий информацию, преобразует модулированный несущий сигнал в двоичное число. Биты поступают в модем через равные промежутки времени. Если модем встречает начало символа, его часы сообщают, когда нужно начать считывать поступающие биты.

Современные модемы передают данные со скоростью от 56 Кбит/с, что обычно соответствует более низкой скорости в бодах. Они сочетают разные технологий для передачи нескольких битов за один бод, модулируя амплитуду, частоту и фазу. Почти все современные модемы являются **дуплексными**, то есть могут передавать информацию в обоих направлениях одновременно, используя различные частоты. Модемы и линии связи, которые не могут передавать ин-

формацию в обоих направлениях одновременно (как одноколейная железная дорога), называются **полудуплексными**. Линии связи, которые могут передавать информацию только в одном направлении, называются **симплексными**.

Цифровые абонентские линии

Взяв однажды планку в 56 Кбит/с, инженеры телефонных компаний с чувством выполненного долга успокоились на достигнутом. Тем временем поставщики услуг кабельного телевидения стали предлагать абонентам подключение к Интернету по общим кабелям на скорости до 10 Мбит/с. Поставщики услуг спутниковой связи пошли еще дальше, обеспечив возможность подключения на скорости свыше 50 Мбит/с. Чем большее значение приобретали услуги по предоставлению доступа в Интернет для телефонных компаний, тем отчетливее они понимали, что для сохранения конкурентоспособности нужно предложить рынку какую-то более совершенную услугу, нежели подключение по обычному модему. В результате этих раздумий на свет появилась новая цифровая услуга доступа в Интернет. Услуги, в которых предлагается пропускная способность, превышающая аналогичный показатель для стандартного модемного соединения, иногда называют **широкополосными**, но, честно говоря, это скорее маркетинговый, чем содержательный технический термин. Он подразумевает наличие нескольких сигнальных каналов, тогда как в узкополосном соединении такой канал только один. Таким образом, теоретически 10-гигабитное соединение Ethernet, работающее гораздо быстрее любого «широкополосного» соединения, широкополосным не является, так как в нем сигнальный канал только один.

Первоначально было предложено несколько технологий доступа под общим именем *x***DSL** (Digital Subscriber Line — **цифровая абонентская линия**) с переменным значением *x*. Далее мы обсудим самую распространенную из них — **ADSL** (Asymmetric DSL — **асимметричная цифровая абонентская линия**). Работы над ADSL все еще продолжаются, и далеко не все стандарты на эту технологию прописаны, поэтому некоторые детали со временем могут корректироваться. Впрочем, общая картина, скорее всего, останется неизменной. За дополнительными сведениями по ADSL обращайтесь к дополнительной литературе [Summers, 1999, Vetter et al., 2000].

Почему обычные модемы работают так медленно? Да потому, что телефоны были изначально предназначены для передачи голоса, и именно с учетом этой цели сформировалась вся система телефонной связи. Передаче данных по телефонным проводам уделялось слишком мало внимания. Пропускная способность провода (он же — **абонентский канал**), связывающего абонентов с автоматической телефонной станцией (АТС), традиционно ограничивалась специальным фильтром. Фактическая пропускная способность абонентского канала во многом зависит от его протяженности, но чаще всего (если протяженность не превышает нескольких километров) достигает 1,1 МГц.

Наиболее распространенная схема предоставления услуг ADSL изображена на рис. 2.35. Ее содержание сводится к удалению фильтра и разделению освободившегося спектра 1,1 МГц на 256 автономных каналов, по 4312,5 Гц каждый. Канал 0 выделяется для голосовых данных. Каналы 1–5 не используются, за счет чего устраняются взаимные помехи сигналов передачи голоса и данных. Из оставшихся 250 каналов два выделяются для восходящей и нисходящей передачи

управляющих сигналов. По остальным каналам передаются пользовательские данные. Таким образом, один ADSL-модем равноценен 250 обычным модемам.

Рис. 2.35. Функционирование ADSL

В принципе, по каждому из оставшихся каналов можно пустить дуплексный поток данных, однако вспомогательные гармоники, перекрестные помехи и другие физические эффекты не позволяют довести фактическую реализацию технологии до теоретического уровня. Решение о том, в какой пропорции разделить каналы на нисходящие и восходящие потоки, принимает поставщик услуги. Технически возможно равное распределение этих каналов, но в большинстве случаев 80–90 % выделяются на организацию нисходящего потока (обычно 32 канала выделяют для восходящего потока, а остальная часть — для нисходящего), поскольку большинство пользователей принимают значительно больше данных, чем отправляют. Именно по этой причине технология ADSL так успешна.

Качество передачи данных по каждому каналу постоянно отслеживается и при необходимости корректируется, поэтому по скоростям каналы могут различаться. Данные передаются в объеме до 15 бит/бод за счет сочетания амплитудной и фазовой модуляции. Если, скажем, для передачи данных доступно 224 нисходящих канала, а скорость передачи сигнала равна 4000 бод при 15 бит/бод, совокупная пропускная способность нисходящего потока составляет 13,44 Мбит/с. На практике соотношение «сигнал/шум» не позволяет приблизиться к такому уровню, но при небольшой удаленности от поставщика услуг и высоком качестве канала скорость 4–8 Мбит/с вполне достижима.

Стандартная конфигурация оборудования ADSL изображена на рис. 2.36. Согласно этой схеме, в помещении пользователя устанавливается **сетевое интерфейсное устройство** (Network Interface Device, **NID**). Эта небольшая пластиковая коробочка символизирует границу между собственностью пользователя и собственностью телефонной компании. Рядом с NID (а иногда и в одном корпусе с этим устройством) устанавливается **сплиттер** (разветвитель) — аналоговый фильтр, разделяющий данные и сигналы на частоте 0–4000 Гц, применяемые для передачи голоса. Поток данных направляется к ADSL-модему, а голосовые сигналы — к телефону. ADSL-модем представляет собой процессор цифровых сигналов, эмулирующий параллельную работу 250 обычных модемов на разных частотах. Поскольку большинство ADSL-модемов выпускаются во внешнем исполнении, их соединение с компьютером должно быть достаточно скоростным. Обычно это требование удовлетворяется путем установки в компьютер платы Ethernet и организации двухзвенного Ethernet-соединения с ADSL-модемом. (Ethernet — распространенный и весьма доступный стандарт организации

локальных сетей.) Иногда ADSL-модем подключается к компьютеру через USB-порт. В будущем следует ожидать появления специальных плат для соединения с ADSL-модемом.

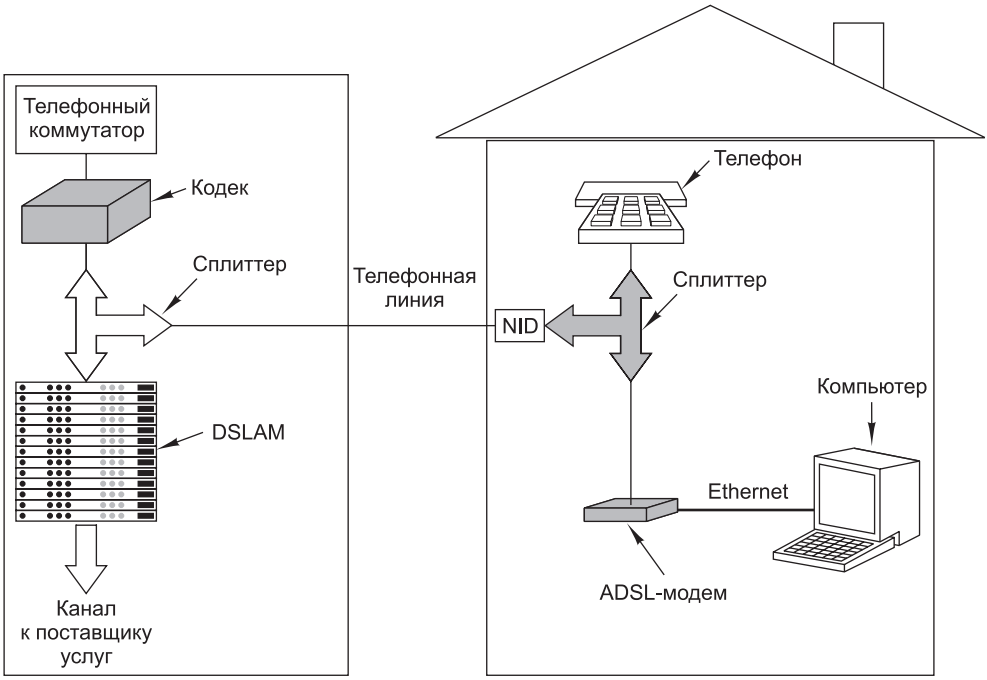


Рис. 2.36. Стандартная конфигурация оборудования ADSL

На противоположной стороне абонентского канала устанавливается другой сплиттер, который отделяет голосовые сигналы и перенаправляет их на обычный телефонный коммутатор. Сигналы с частотой свыше 26 кГц передаются **мультиплексору доступа к цифровой абонентской линии** (Digital Subscriber Line Access Multiplexer, **DSLAM**). После преобразования цифровых сигналов в поток битов происходит формирование пакетов, которые затем направляются поставщику услуг.

Кабельный Интернет

Многие телевизионные компании предлагают пользователям возможность доступа в Интернет по кабельным сетям. Эта технология несколько отличается от ADSL, поэтому ее стоит рассмотреть отдельно. Во владении каждого оператора кабельного телевидения, помимо центрального офиса, есть ряд головных узлов (помещений с электронным оборудованием), рассредоточенных по территории города. К центральному офису все они подключены широкополосным или оптоволоконным кабелем.

От каждого головного узла к конечным потребителям отходит один или несколько кабелей. Чтобы к такому кабелю можно было подключиться, он должен проходить рядом с помещениями, в которых находятся пользователи. При этом к одному и тому же кабелю подключаются сотни пользователей. Как правило,

пропускная способность такого кабеля составляет около 750 МГц. Как видно, концептуальное отличие кабельного доступа от технологии ADSL заключается в отсутствии индивидуального канала, подведенного к офису поставщика услуг. Впрочем, на практике выгоды от наличия собственного канала пропускной способностью 1,1 МГц, с одной стороны, и общего с еще четырьмя сотнями пользователей канала совокупной пропускной способностью 200 МГц, с другой, примерно равноценны (объясняется это тем, что в каждый отдельно взятый момент времени из 400 пользователей в сети находятся не более половины). Более того — глубокой ночью кабельный Интернет работает значительно быстрее, чем днем, в то время как скорость передачи данных по каналу ADSL в течение суток одинакова. Логика такова: чтобы получить оптимальный доступ в Интернет по кабельному каналу, нужно жить либо в очень богатом районе (где дома находятся на большом расстоянии друг от друга, а, следовательно, к одному кабелю подключено не так уж много пользователей), либо в очень бедном (где никто не может себе позволить приобрести такую услугу).

Поскольку к одному кабелю подключаются многочисленные пользователи, актуальной проблемой является временная и частотная регламентация потребления пропускной способности. Чтобы понять, как эта проблема решается, придется сделать небольшой экскурс в технологию кабельного телевидения. В США для вещания кабельных каналов выделен частотный диапазон 54–550 МГц (из него, правда, исключается диапазон 88–108 МГц, предназначенный для FM-радиостанций). Каждый канал занимает 6 МГц (включая защитные полосы, предотвращающие взаимные помехи смежных каналов). В Европе нижний порог кабельного диапазона — 65 МГц, а каналы занимают по 6–8 МГц (за счет этого обеспечивается повышенное разрешение по стандартам PAL/SECAM); во всем остальном схема распределения частот аналогична американской. В обоих случаях нижняя часть диапазона не используется для передачи телевизионных сигналов.

Пытаясь реализовать технологию доступа в Интернет по кабелю, операторы столкнулись с двумя проблемами:

1. Как предотвратить помехи при одновременной передаче данных и телевизионного сигнала?
2. Как организовать двунаправленный трафик при однонаправленных усилителях?

Выбранные решения таковы. Современные кабели работают на частоте значительно выше 550 МГц, достигая 750 МГц и более. Восходящие (то есть направленные от пользователя к головному узлу) каналы занимают диапазон 5–42 МГц (в Европе он чуть выше), в то время как для передачи нисходящего (от головного узла к пользователю) трафика используются высокие частоты (рис. 2.37).

Обратите внимание: поскольку телевизионные сигналы передаются исключительно в нисходящем направлении, восходящие усилители могу работать только в диапазоне 5–42 МГц, а нисходящие — в диапазоне от 54 МГц и выше. Таким образом, пропускная способность двух направлений оказывается асимметричной, поскольку восходящий диапазон значительно меньше нисходящего. Впрочем, это обстоятельство не сильно беспокоит операторов кабельного телевидения, так как и трафик по большей части передается *к* пользователю, а не *от* него. В конце

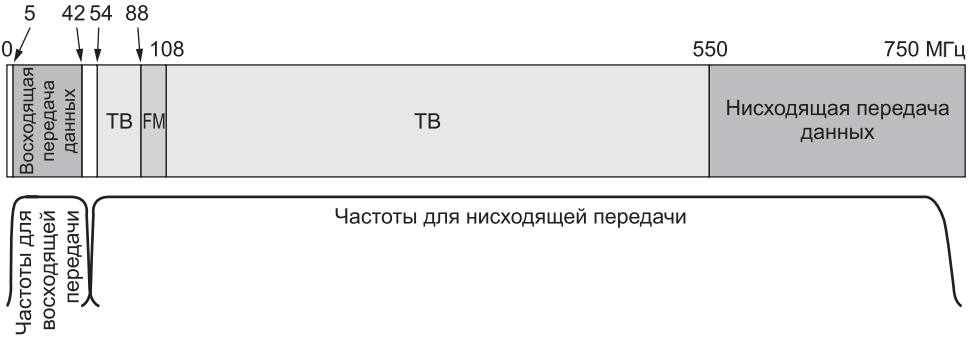


Рис. 2.37. Распределение частот в стандартной системе кабельного телевидения с возможностью доступа в Интернет

концов, телефонные компании тоже успешно предоставляют услугу DSL с асимметричным доступом, хотя никаких технических ограничений на восходящий трафик здесь не существует.

Для подключения к Интернету конечного пользователя применяются кабельные модемы. Это устройства с двумя интерфейсами — для подключения к компьютеру, с одной стороны, и к кабельной сети, с другой. Интерфейс «компьютер–кабельный модем» несложен — как и в случае с ADSL, для передачи данных организуется миниатюрная сеть Ethernet. В будущем кабельные модемы, скорее всего, будут производиться в виде плат, устанавливаемых в системный блок, — точно так же, как это произошло со старыми телефонными модемами.

На противоположной стороне устанавливается более сложное оборудование. Стандарты кабельных соединений — это тема из области радиотехники, поэтому в нашем контексте ее подробное изложение неуместно. Единственное, что стоит отметить, так это беспрерывную работу кабельных модемов — в этом отношении они напоминают ADSL-модемы. Соединение устанавливается и поддерживается постоянно и прерывается только при отключении источника питания — связано это с тем, что операторы кабельных сетей не взимают повременную плату за свои услуги.

Чтобы лучше понять, как работают кабельные модемы, рассмотрим последовательность операций при подсоединении и включении модема. Сначала модем просматривает содержимое нисходящих каналов в поисках специального пакета, который с определенной регулярностью отправляется с головного узла и содержит системные параметры для недавно подключенных модемов. Обнаружив таковой, модем объявляет о своем присутствии в одном из восходящих каналов. Далее головной узел назначает модему определенные восходящие и нисходящие каналы. Впоследствии, если головной узел сочтет необходимым сбалансировать нагрузку, модему могут быть назначены другие каналы.

Затем модем определяет расстояние до головного узла путем отправки ему специального пакета и вычисления времени ответа. Этот процесс называется **калибровкой** (ranging). Зная расстояние до головного узла, модем может соответствующим образом скорректировать работу восходящих каналов. Дело в том, что восходящий поток данных подразделяется на временны́е интервалы, или **мини-слоты** (minislots). Каждый восходящий пакет должен уместиться в рамках

одного или нескольких последовательных мини-слотов. Головной узел регулярно высылает оповещения о начале новых циклов мини-слотов, но, так как модемы находятся на разных расстояниях от головного узла, они получают эти оповещения в разное время. В то же время, зная, на каком расстоянии от головного узла он находится, модем может вычислить фактическое время начала мини-слота. Длина мини-слота определяется характеристиками конкретной сети. Полезная нагрузка одного мини-слота обычно составляет 8 байт.

В ходе инициализации головной узел привязывает каждый модем к определенному мини-слоту, в результате тот получает возможность отравлять запросы на предоставление пропускной способности. Обычно к одному и тому же мини-слоту привязываются несколько модемов, за счет чего формируется механизм конкуренции. Перед отправкой пакета с компьютера в сеть он передается модему, который затем запрашивает соответствующее количество мини-слотов. Если запрос удовлетворяется, головной узел отсылает по нисходящему каналу подтверждение, в котором указывает зарезервированные для передачи пакета мини-слоты. Далее, начиная с первого зарезервированного мини-слота, начинается отправка. Запросы на передачу дополнительных пакетов встраиваются в специальное поле заголовка.

Если в условиях конкуренции за запрошенный мини-слот модем не получает подтверждение, он ждет случайный интервал времени и повторяет запрос. С каждой неуспешной попыткой время ожидания удваивается, что способствует разряжению интенсивного трафика.

Нисходящие каналы управляются по-другому. Во-первых, при нисходящей передаче отправитель всего один — головной узел. Следовательно, состязательность отсутствует, равно как и необходимость выделения мини-слотов, которые, по существу, есть не что иное, как средство статистического мультиплексирования с разделением времени. Во-вторых, нисходящий трафик обычно значительно интенсивнее, чем восходящий, поэтому он передается в пакетах по 204 байта. В состав пакета, помимо полезной нагрузки в 184 байта, входит код исправления ошибок Рида–Соломона и некоторые другие служебные поля. Этот размер пакета выбран в целях совместимости с цифровым телевидением формата MPEG-2 — в итоге каналы нисходящей передачи телевизионного сигнала и данных форматируются единообразно. Логическая схема этих соединений изображена на рис. 2.38.

Впрочем, вернемся к процедуре инициализации модема. После калибровки, получения восходящего и нисходящего каналов и назначения минислотов модем может приступать к передаче пакетов. Пакеты отправляются на головной узел, с которого они по выделенному каналу уходят в центральный офис оператора кабельного телевидения, а от него — к поставщику услуг Интернета (Internet Service Provider, ISP), в качестве которого может выступать и сам оператор. Первый пакет, отправляемый поставщику услуг, содержит запрос на предоставление в динамическом режиме сетевого адреса (IP-адреса). Другой запрос в составе этого пакета касается точного времени дня.

На следующем этапе решаются вопросы безопасности. По одному кабелю свои данные передают множество пользователей, поэтому при большом желании пользователь может организовать перехват всего проходящего трафика. Чтобы не допустить со стороны соседей коллективного слежения друг за другом, весь трафик, в каком бы направлении он ни отправлялся, в обязательном порядке

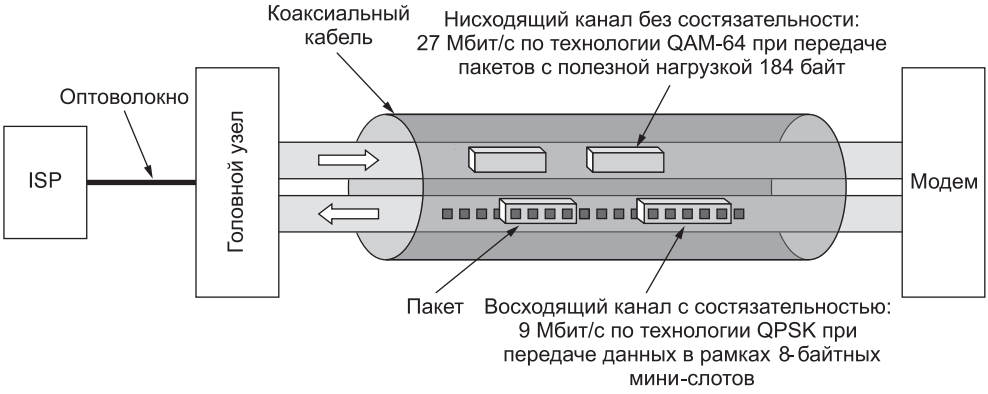


Рис. 2.38. Стандартная организация восходящих и нисходящих каналов в США. Технология QAM-64 (квадратурная амплитудная модуляция) допускает передачу со скоростью 6 бит/Гц, но работает только на высоких частотах. Технология QPSK (квадратурная фазовая модуляция) действует на низких частотах, зато максимальная скорость передачи составляет 2 бит/Гц

шифруется. Поэтому в процессе инициализации, помимо прочего, определяются ключи шифрования. Казалось бы, провести согласование секретного ключа между модемом и головным узлом под бдительным оком тысяч пользователей невозможно. На самом деле, это не так — для определения ключа шифрования задействуется алгоритм Диффи–Хелмана [Kaufman et al., 2002].

Наконец, модем регистрируется в сети и сообщает по защищенному каналу свой уникальный идентификатор. На этом процесс инициализации заканчивается — пользователь может начинать работу.

Наш обзор кабельных модемов получился довольно сжатым. За подробностями обратитесь к дополнительной литературе [Adams and Dulchinos, 2001; Donaldson and Jones, 2001; Dutta-Roy, 2001].

Цифровые фотокамеры

Все больше компьютерные технологии внедряются в сферу цифровой фотографии — уже сейчас цифровые фотокамеры вполне правомерно рассматривать как один из видов компьютерных периферийных устройств. Давайте вкратце рассмотрим принцип их работы. Все камеры снабжены объективом, с помощью которого в задней части камеры формируется изображение объекта. В традиционной камере в качестве носителя скрытых изображений, которые формируются в момент проникновения света, выступает фотопленка. Изображения проявляются в лаборатории за счет воздействия определенных химических реактивов. Принцип действия цифровой камеры аналогичен за одним исключением — вместо пленки носителем изображения становится прямоугольная матрица светочувствительных **устройств с зарядовой связью** (Charge-Coupled Devices, **CCD**). (Некоторые цифровые камеры действуют на основе технологии КМОП, но вариант с CCD более распространен.)

При попадании на устройство CCD света устройство получает электрический заряд. Чем больше света, тем существеннее изменение заряда. Заряд считывается

аналогово-цифровым преобразователем в виде целого числа от 0 до 255 (в камерах низкой ценовой категории) или от 0 до 4095 (на цифровых однообъективных зеркальных фотоаппаратах). Соответствующая схема изображена на рис. 2.39.

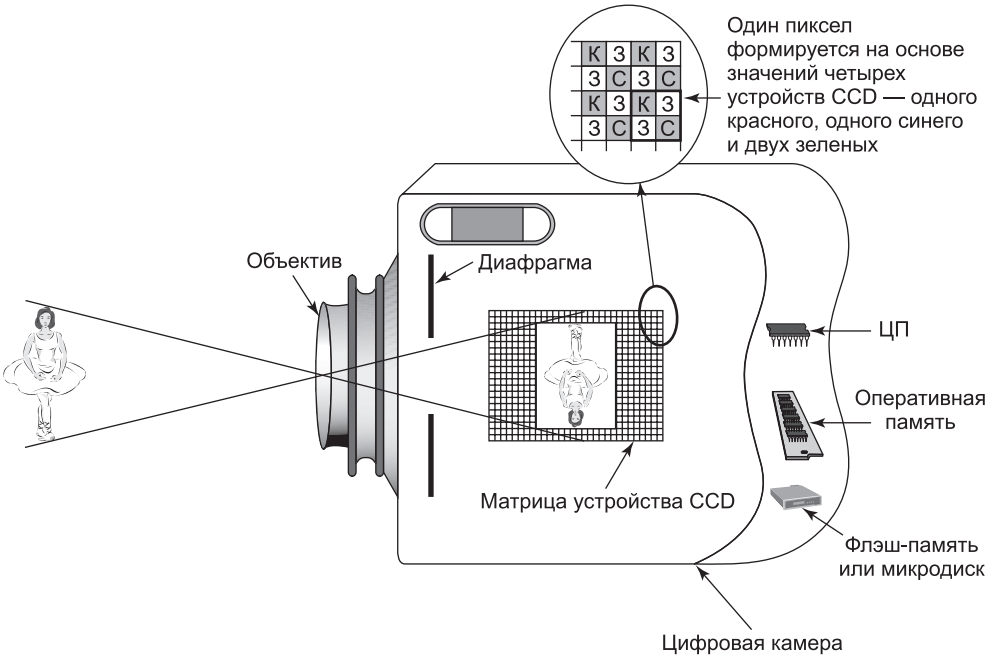


Рис. 2.39. Цифровая камера

Каждое устройство CCD, независимо от падающего на него света, на выходе генерирует единственное значение. Для формирования цветных изображений устройства CCD объединяются в группы из четырех элементов. Поверх группы размещается **фильтр Байера** (Bayer filter), который делает одно устройство CCD чувствительным к красному цвету, другое — к синему, а два оставшихся — к зеленому. Наличие двух зеленых элементов объясняется двумя факторами: во-первых, это удобнее, а во-вторых, человеческий глаз воспринимает зеленый цвет лучше, чем синий и красный. Если производитель цифровой камеры заявляет, что ее разрешение равно 6 млн пикселов, знайте — это неправда. В ней 6 млн устройств CCD, которые в совокупности формируют 1,5 млн пикселов. При таком разрешении изображение считывается в виде матрицы 2828 × 2121 (в недорогих камерах) или 3000 × 2000 (в однообъективных зеркальных фотоаппаратах) пикселов. Дополнительные пикселы генерируются путем программной интерполяции.

При нажатии кнопки открытия затвора объектива программное обеспечение камеры выполняет три операции: устанавливает фокус, определяет экспозицию и проводит балансировку белого. Автоматическая фокусировка осуществляется путем анализа высокочастотных данных изображения и выдвижения объектива на предельную позицию в целях максимальной детализации. При определении экспозиции сначала вычисляется интенсивность света, падающего на CCD, после диафрагма и выдержка корректируются таким образом, чтобы полученное

значение интенсивности пришлось на середину диапазона CCD. Балансировка белого сводится к измерению спектра падающего света с целью последующей цветокоррекции.

Далее изображение считывается с CCD и сохраняется в виде матрицы пикселов во встроенной оперативной памяти камеры. Профессиональные однообъективные зеркальные фотоаппараты, с которыми работают фотокорреспонденты, могут в течение пяти секунд снимать по восемь кадров с высоким разрешением в секунду; при этом объем встроенной оперативной памяти, в которой изображения размещаются перед последующей обработкой и постоянным хранением, составляет около 1 Гбайт. В недорогих камерах оперативной памяти меньше, но все равно вполне достаточно.

После создания снимка программное обеспечение проводит цветокоррекцию на основе баланса белого, тем самым нейтрализуя избыток красного или синего света (что имеет место, например, при фотографировании объекта, находящегося в тени, а также при использовании вспышки). Затем выполняются алгоритмы шумоподавления и корректировки дефектных устройств CCD. После этого (если соответствующая функция включена) производится попытка повысить резкость изображения — выполняется поиск краев и увеличение интенсивности градиента вокруг них.

Наконец, изображение сжимается с целью уменьшения объема занимаемой им памяти. Самый распространенный формат, применяемый для этих целей, — **JPEG** (Joint Photographic Experts Group — объединенная группа экспертов в области фотографии). Он предусматривает двухмерное пространственное преобразование Фурье и удаление высокочастотных составляющих. Конечное изображение оказывается весьма компактным, но мелкие детали утрачиваются.

По окончании обработки изображение записывается на постоянный носитель, в качестве которого обычно выступает карта флэш-памяти или небольшой съемный жесткий диск — так называемый **микродиск**. На обработку и запись каждого изображения уходит несколько секунд.

Затем пользователь может подключить камеру к компьютеру — посредством, например, кабеля USB или FireWire. Это позволяет перенести изображения из памяти камеры на жесткий диск компьютера. При помощи специального программного обеспечения (например, редактора Adobe Photoshop) пользователь может обрезать изображение, настроить яркость, контраст и баланс, увеличить резкость или, наоборот, частично размыть изображение, удалить ненужные элементы и наложить в произвольном сочетании фильтры. Удовлетворившись результатом, пользователь волен распечатать изображения на цветном принтере, разместить их в Интернете, а также записать на компакт-диск или DVD для архивации или последующей печати.

По вычислительным мощностям, объему оперативной памяти и дискового пространства, равно как и по сложности программного обеспечения, цифровые однообъективные зеркальные фотоаппараты (Single-Lens Reflex, SLR) сопоставимы с настольными системами двух–трехлетней давности. Помимо вышеперечисленных операций, компьютер такого фотоаппарата должен обеспечивать взаимодействие с процессором объектива и вспышки, обновлять изображение на жидкокристаллическом экране, не говоря уже о координации действий всех кнопок, колесиков, индикаторов, дисплеев и прочих приспособлений в реальном времени.

Коды символов

У каждого компьютера есть набор символов, который он использует. Как минимум, этот набор включает 26 прописных и 26 строчных букв[1], цифры от 0 до 9, а также некоторые специальные символы, в том числе пробел, точку, запятую, минус и символ возврата каретки и т. д.

Для того чтобы передавать эти символы в компьютер, каждому из них приписывается номер, например, a = 1, b = 2, ..., z = 26, + = 27, — = 28. Представление символа в виде целого числа называется **кодом символа**. Важно отметить, что связанные между собой компьютеры должны поддерживать одни и те же коды символов, иначе они не смогут обмениваться информацией. По этой причине были разработаны стандарты. Здесь мы рассмотрим два самых важных из них.

ASCII

Один из двух широко распространенных кодов называется **ASCII** (American Standard Code for Information Interchange — **американский стандартный код для обмена информацией**). Каждый ASCII-символ содержит 7 бит, таким образом, всего можно закодировать 128 символов. Однако, поскольку компьютеры байт ориентированы, каждый ASCII символ обычно хранится в отдельном байте. Таблица 2.4 показывает ASCII коды. Коды от 0 до 1F (в шестнадцатеричной записи) соответствуют управляющим символам, которые не выводятся на печать. Коды от 128 до 255 не входят в кодировку ASCII; на IBM PC за ними были закреплены специальные символы (улыбающиеся лица и т. д.), которые до сих пор поддерживаются большинством компьютеров.

Таблица 2.4. Набор символов ASCII

Число	Название	Значение	Число	Название	Значение
0	NUL	Нуль	8	BS	BackSpace (Отступ назад)
1	SOH	Start Of Heading (Начало заголовка)	9	HT	Horizontal Tab (Горизонтальная табуляция)
2	STX	Start Of Text (Начало текста)	A	LF	Line Feed (Перевод строки)
3	ETX	End of Text (Конец текста)	B	VT	Vertical Tab (Вертикальная табуляция)
4	EOT	End Of Transmission (Конец передачи)	C	FF	From Feed (Перевод страницы)
5	ENQ	ENQuiry (Запрос)	D	CR	Carriage Return (Возврат каретки)
6	ACK	ACKnoligement (Подтверждение приема)	E	SO	Shift Out (Переключение на дополнительный регистр)
7	BEL	Bell (Звуковой сигнал)	F	SI	Shift In (Переключение на стандартный регистр)

[1] В английском языке. — *Примеч. перев.*

Число	Название	Значение	Число	Название	Значение
10	DLE	Data Link Escape (Смена канала данных)	18	CAN	CANcel (Отмена)
11	DC1	Device Control 1 (Управление устройством 1)	19	EM	End of Medium (Конец носителя)
12	DC2	Device Control 2 (Управление устройством 2)	1A	SUB	SUBstitute (Подстрочный индекс)
13	DC3	Device Control 3 (Управление устройством 3)	1B	ESC	ESCape (Выход)
14	DC4	Device Control 4 (Управление устройством 4)	1C	FS	File Separator (Разделитель файлов)
15	NAK	Negative AcKnolidgement (Неподтверждение приема)	1D	GS	Group Separator (Разделитель группы)
16	SYN	SYNcronous idle (Пауза)	1E	RS	Record Separator (Разделитель записи)
17	ETB	End of Transmission Block (Конец блока передачи)	1F	US	Unit Separator (Разделитель модуля)

Число	Символ	Число	Символ	Число	Символ	Число	Символ	Число	Символ	Число	Символ
20	(пробел)	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	Q
22	“	32	2	42	B	52	R	62	b	72	R
23	#	33	3	43	C	53	S	63	c	73	S
24	$	34	4	44	D	54	T	64	d	74	T
25	%	35	5	45	E	55	U	65	e	75	U
26	&	36	6	46	F	56	V	66	f	76	V
27	’	37	7	47	G	57	W	67	g	77	W
28	(	38	8	48	H	58	X	68	h	78	X
29	)	39	9	49	I	59	Y	69	i	79	Y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Многие управляющие ASCII-символы предназначены для передачи данных. Например, послание может состоять из символа начала заголовка SOH (Start of Header), самого заголовка, символа начала текста STX (Start of Text), самого текста, символа конца текста ETX (End of Text) и, наконец, символа конца передачи EOT (End of Transmission). Однако на практике послания, отправляемые по телефонным линиям и сетям, форматируются по-другому, так что управляющие ASCII-символы для передачи практически не используются.

Печатные ASCII-символы включают буквы верхнего и нижнего регистров, цифры, знаки пунктуации и некоторые математические символы.

Unicode

Компьютерная промышленность развивалась преимущественно в США, что привело к появлению кода ASCII, более подходящего для английского языка, чем для других языков. Во французском языке есть надстрочные знаки, в немецком — умляуты и т. д. В некоторых европейских языках есть несколько букв, которых нет в наборе ASCII-символов. Некоторые языки имеют совершенно другой алфавит (например, русский или арабский), а у некоторых вообще нет алфавита (например, китайский). Компьютеры распространились по всему свету, и поставщики программного обеспечения хотят реализовывать свою продукцию не только в англоязычных, но и в тех странах, где большинство пользователей не говорят по-английски и нужен другой набор символов.

Первой попыткой расширения кода ASCII стал стандарт IS 646, который добавлял к набору ASCII-символов еще 128 символов, в результате чего получился 8-разрядный набор под названием **Latin-1**. Добавлены были в основном латинские буквы со штрихами и диакритическими знаками. Следующей попыткой был стандарт IS 8859, который ввел понятие **кодовой страницы**. Кодовая страница — набор из 256 символов для определенного языка или группы языков, в IS 8859-1 это набор Latin-1. Стандарт IS 8859-2 включает славянские языки с латинским алфавитом (например, чешский, польский и венгерский), стандарт IS 8859-3 описывает символы турецкого, мальтийского и галисийского языков, эсперанто и т. д. Главным недостатком такого подхода является то, что программное обеспечение должно контролировать, с какой именно кодовой страницей оно имеет дело, при этом смешивать языки недопустимо. К тому же эта система не охватывает японский и китайский языки.

Группа компьютерных компаний разрешила эту проблему, создав новую систему кодирования под названием **Unicode**, и объявила эту систему международным стандартом (IS 10646). Unicode поддерживается некоторыми языками программирования (например, Java), некоторыми операционными системами (например, Windows NT) и многими приложениями. Вероятно, эта система будет распространяться по всему миру.

Основная идея Unicode — приписать каждому символу единственное постоянное 16-разрядное значение, которое называется **кодовым пунктом**. Многобайтные символы и символы-заменители не используются. Поскольку каждый символ состоит из 16 бит, писать программное обеспечение гораздо проще.

Так как Unicode-символы состоят из 16 бит, всего получается 65 536 кодовых пунктов. Поскольку во всех языках мира в общей сложности около 200 000 символов, кодовые пункты являются очень дефицитным ресурсом, который нужно

распределять с большой осторожностью. Около половины кодов уже распределено, и консорциум, разработавший Unicode, постоянно рассматривает предложения по распределению оставшейся части. Чтобы ускорить принятие системы Unicode, консорциум использовал набор Latin-1 для кодов от 0 до 255, обеспечивающий простое преобразование ASCII- в Unicode-символы. Во избежание излишней растраты кодов каждый диакритический знак имеет собственный код, а объединение диакритического знака с той или иной буквой должно осуществляться программным обеспечением.

Вся совокупность кодов разделена на блоки, каждый блок содержит 16 кодов. Каждый алфавит в Unicode имеет ряд последовательных зон. Приведем некоторые примеры (в скобках указано число задействованных кодов): латынь (336), греческий (144), русский (256), армянский (96), иврит (112), деванагари (128), гурмукхи (128), ория (128), телугу (128) и каннада (128). Отметим, что каждому из этих языков приписано больше кодов, чем в нем есть букв. Это сделано отчасти потому, что во многих языках у каждой буквы есть несколько вариантов написания. Например, каждая буква в английском языке представлена в двух вариантах: там есть строчные и ПРОПИСНЫЕ буквы. В некоторых языках буквы имеют три или более формы написания, выбор конкретного варианта зависит от того, где находится буква: в начале, конце или середине слова.

Кроме того, некоторые коды были приписаны диакритическим знакам (112), знакам пунктуации (112), подстрочным и надстрочным знакам (48), знакам валют (48), математическим символам (256), геометрическим фигурам (96), рисункам (192).

Нужны также символы для китайского, японского и корейского языков. Сначала идут 1024 фонетических символа (например, катакана и бопомофо), затем иероглифы, используемые в китайском и японском языках (20 992), потом слоги корейской азбуки хангыль (11 156).

Чтобы пользователи могли создавать новые символы для особых целей, существуют еще 6400 кодов.

Хотя система Unicode разрешила многие проблемы, связанные с интернационализацией, она все же не позволила разрешить абсолютно все проблемы. Например, латинский алфавит упорядочен, а иероглифы — нет, поэтому программа для английского языка может расположить слова «cat» и «dog» по алфавиту, сравнив значение кодов первых букв, а программе для японского языка нужны дополнительные таблицы, чтобы можно было вычислять, в каком порядке расположены символы в словаре.

Еще одна проблема состоит в том, что постоянно появляются новые слова. 50 лет назад никто не говорил об апплетах, киберпространстве, гигабайтах, лазерах, модемах, смайликах или видеопленках. Появление новых слов в английском языке новых кодов не требует, а вот в японском они нужны. Помимо новых терминов, необходимо также добавить по крайней мере 20 000 новых имен собственных и географических названий (в основном китайских). Шрифт Брайля, вероятно, тоже должен быть задействован. В наличии тех или иных особых символов заинтересованы и представители различных профессиональных сообществ. Консорциум Unicode рассматривает все новые предложения и выносит по ним решения.

В системе Unicode используется один и тот же код для символов, которые выглядят почти одинаково, но имеют несколько значений или пишутся немного по-разному в китайском и японском языках (как если бы английские текстовые процессоры всегда писали слово «blue» как «blew», потому что они произносятся одинаково). Одни считают такой подход оптимальным для экономии скудного запаса кодов, другие рассматривают его как англосаксонский культурный империализм (а вы думали, что назначение символам 16-разрядных кодов не носит политического характера?). Дело усложняется тем, что полный японский словарь содержит 50 000 иероглифических знаков (не считая знаков, используемых только в именах собственных), поэтому при наличии 20 992 кодов приходится делать выбор и чем-то жертвовать. Далеко не все японцы считают, что консорциум компьютерных компаний, даже если некоторые из них японские, идеально подходит для принятия решений о том, чем именно нужно жертвовать.

В общем, оказалось, что 65 536 кодовых пунктов недостаточно для всех потребностей, поэтому в 1996 году были определены дополнительные шестнадцать 16-разрядных **плоскостей**, в результате чего общее количество символов увеличилось до 1 114 112.

UTF-8

Стандарт Unicode был лучше ASCII, но со временем в нем тоже возникла нехватка кодовых пунктов. Кроме того, он требовал 16 бит для представления «чистого» ASCII-текста, что было расточительно. Для решения этих проблем была разработана новая схема кодирования **UTF-8 UCS Transformation Format**. Сокращение UCS в названии означает «Universal Character Set» — по сути это Unicode. Коды UTF-8 имеют переменную длину от 1 до 4 байт, и позволяют кодировать до двух миллиардов символов. Этот способ кодировки сейчас доминирует в World Wide Web.

Одно из преимуществ UTF-8 заключается в том, что коды от 0 до 127 используются для представления ASCII-символов. Таким образом, одному символу соответствует один байт (вместо двух байтов в Unicode). Для символов, не входящих в набор ASCII, старший бит первого байта устанавливается в 1; это означает, что за ним следуют один и более дополнительных байтов. Всего используются шесть разных форматов, представленных в табл. 2.5. Биты с пометкой «d» являются битами данных.

Таблица 2.5. Схема кодирования UTF-8

Биты	Байт 1	Байт 2	Байт 3	Байт 4	Байт 5	Байт 6
7	0ddddddd					
11	110ddddd	10dddddd				
16	1110dddd	10dddddd	10dddddd			
21	11110ddd	10dddddd	10dddddd	10dddddd		
26	111110dd	10dddddd	10dddddd	10dddddd	10dddddd	
31	1111110x	10dddddd	10dddddd	10dddddd	10dddddd	10dddddd

UTF-8 обладает рядом преимуществ перед Unicode и другими схемами. Прежде всего, если в программе или документе используются только символы

из набора ASCII, каждый символ может быть представлен 8 битами. Во-вторых, первый байт каждого символа UTF-8 однозначно определяет порядок байтов в символе. В-третьих, дополнительные байты символа UTF-8 всегда начинаются с префикса 10, который никогда не встречается в начальном байте; соответственно код становится самосинхронизирующимся. В частности, при возникновении ошибки передачи данных или памяти всегда возможно перейти к началу следующего (неповрежденного) символа.

Обычно UTF-8 используется для кодирования только 17 плоскостей Unicode, хотя схема позволяет представить много более 1 114 112 кодовых пунктов. Но если антропологи найдут где-нибудь в Новой Гвинее племя с неизвестным языком (или мы вступим в контакт с инопланетными цивилизациями), UTF-8 успешно справится с представлением их алфавитов или идеограмм.

Краткое содержание главы

Компьютерные системы состоят из трех компонентов: процессоров, памяти и устройств ввода-вывода. Назначение процессора заключается в том, чтобы последовательно вызывать команды из памяти, декодировать и выполнять их. Цикл выборка — декодирование — выполнение всегда можно представить в виде алгоритма. Выборка, декодирование и выполнение команд определенной программы иногда выполняется программой-интерпретатором, работающей на более низком уровне. Для повышения быстродействия во многих компьютерах имеется один или несколько конвейеров или поддерживается суперскалярная архитектура с несколькими функциональными блоками, которые действуют параллельно. Конвейер позволяет разбивать команды на отдельные шаги, с одновременным выполнением шагов разных команд.

Широко распространены системы с несколькими процессорами. Компьютеры с параллельной обработкой включают матричные процессоры, в которых одна и та же операция выполняется одновременно над разными наборами данных; мультипроцессоры, в которых несколько процессоров разделяют общую память; и мультикомпьютеры, в которых у каждого компьютера есть собственная память, но при этом компьютеры связаны между собой и пересылают друг другу сообщения.

Память можно разделить на основную и вспомогательную. Основная память используется для хранения программ, которые выполняются в данный момент. Время доступа к основной памяти невелико (максимум несколько десятков наносекунд) и не зависит от адреса, к которому происходит обращение. Кэш-память еще больше сокращает время доступа. Память может быть оснащена кодом исправления ошибок для повышения надежности.

Время доступа к вспомогательной памяти, напротив, гораздо больше (от нескольких миллисекунд и выше) и зависит от расположения считываемых и записываемых данных. Наиболее распространенные виды вспомогательной памяти — магнитные ленты, магнитные диски, оптические диски. Магнитные диски существуют в нескольких вариантах: дискеты, винчестеры, IDE-диски, SCSI-диски и RAID-массивы. Среди оптических дисков можно назвать диски CD-ROM, CD-R, DVD и Blu-ray.

Устройства ввода-вывода используются для передачи информации в компьютер и из компьютера. Они связаны с процессором и памятью одной или несколькими шинами. В качестве примеров можно назвать терминалы, мыши, принтеры и модемы. Большинство устройств ввода-вывода используют код ASCII, хотя Unicode уже стремительно распространяется по всему миру, а UTF-8 получает все большее распространение по мере того, как компьютерная отрасль все больше ориентируется на Web.

Вопросы и задания

1. Рассмотрим машину с трактом данных, который изображен на рис. 2.2. Предположим, что загрузка регистров АЛУ занимает 5 нс, работа АЛУ — 10 нс, а помещение результата обратно в регистр — 5 нс. Сколько миллионов команд в секунду максимум способна выполнять эта машина при отсутствии конвейера?
2. Зачем нужен шаг 2 в списке шагов, приведенном в подразделе «Выполнение команд» раздела «Процессоры»? Что произойдет, если этот шаг пропустить?
3. На компьютере 1 выполнение каждой команды занимает 10 нс, а на компьютере 2 — 5 нс. Можете ли вы с уверенностью сказать, что компьютер 2 работает быстрее? Аргументируйте ответ.
4. Предположим, что вы разрабатываете компьютер на одной микросхеме для использования во встроенных системах. Вся память находится на микросхеме и работает с той же скоростью, что и центральный процессор. Рассмотрите принципы, изложенные в подразделе «Принципы проектирования современных компьютеров» раздела «Процессоры», и скажите, важны ли они в данном случае (высокая производительность желательна).
5. Чтобы конкурировать с недавно изобретенным печатным станком, один средневековый монастырь решил наладить массовое производство рукописных книг. Для этого в большом зале собралось огромное количество писцов. Настоятель монастыря называл первое слово книги, и все писцы записывали его. Затем настоятель называл второе слово, и все писцы записывали его. Этот процесс повторялся до тех пор, пока не была прочитана вслух и переписана вся книга. На какую из систем параллельной обработки информации, перечисленных в подразделе «Параллелизм на уровне процессоров» раздела «Процессоры», эта система больше всего похожа?
6. При продвижении сверху вниз по пятиуровневой иерархической структуре памяти время доступа возрастает. Каково соотношение времени доступа к оптическому диску и к регистровой памяти? (Предполагается, что диск уже вставлен.)
7. Устраивая стандартный опрос (например, «Верите ли вы, что Дед Мороз существует?»), социологи рассчитывают получить от респондентов один из трех ответов: «да», «нет» или «не могу ничего сказать по этому поводу». Предположим, что имея в виду это обстоятельство, компания Socimagnetic Computer решила сконструировать компьютер для обработки данных опросов.

В этом компьютере реализована троичная память; иными словами, каждый байт (точнее, «трайт») состоит из 8 трит, а каждый трит может принимать значение 0, 1 или 2. Сколько тритов необходимо для хранения 6-разрядного числа? Составьте формулу вычисления количества тритов, необходимых для хранения n бит.

8. Сосчитайте скорость передачи данных в человеческом глазу, используя следующую информацию. Поле зрения состоит приблизительно из 10^6 элементов (пикселов). Каждый пиксел может образовываться наложением трех основных цветов, каждый из которых имеет 64 степени интенсивности. Временно́е разрешение 100 миллисекунд.
9. Сосчитайте скорость передачи данных в человеческом ухе, исходя из следующих данных. Человек слышит звуки на частоте до 22 кГц. Чтобы определить всю информацию, содержащуюся в звуковом сигнале частотой 22 кГц, нужно провести его дискретизацию на удвоенной частоте — 44 кГц. 16-разрядной дискретизации вполне достаточно для фиксации всех слуховых данных (то есть ухо различает не более 65 535 уровней интенсивности звука).
10. Генетическая информация у всех живых существ кодируется в молекулах ДНК. Молекула ДНК представляет собой линейную последовательность четырех основных нуклеотидов: A, C, G и T. Геном человека содержит приблизительно 3×10^9 нуклеотидов в форме 30 000 генов. Какова общая информационная емкость человеческого генома (в битах)? Какова средняя информационная емкость гена (в битах)?
11. Компьютер может содержать 1 073 741 824 байт памяти. Почему разработчики выбрали такое странное число вместо какого-нибудь хорошо запоминающегося, например, 1 000 000 000?
12. Придумайте 7-разрядный код Хэмминга с битами четности для чисел от 0 до 9.
13. Придумайте код для чисел от 0 до 9 с интервалом Хэмминга, равным 2.
14. В коде Хэмминга некоторые биты «теряются» в том смысле, что они используются для проверки и не несут никакой информации. Какой процент пустых битов содержится в посланиях, полная длина которых (данные плюс биты проверки) составляет $2^n - 1$? Сосчитайте значение этого выражения при n от 3 до 10.
15. Расширенные ASCII-символы представляются 8 битами. Соответствующая кодировка каждого символа по Хэммингу может быть представлена цепочкой из трех шестнадцатеричных цифр. Закодируйте следующий расширенный ASCII-текст из пяти символов с использованием кода Хэмминга с контролем четности: «Earth». Представьте свой ответ в виде цепочки шестнадцатеричных цифр.
16. Следующая цепочка шестнадцатеричных цифр кодирует расширенные ASCII-символы с использованием кода Хэмминга с контролем четности: 0D3 DD3 0F2 5C1 1C5 CE3. Декодируйте эту строку и запишите исходные символы.
17. Диск, изображенный на рис. 2.16, имеет 1024 сектора на дорожке и скорость вращения 7200 оборотов в минуту. Какова скорость передачи данных на одной дорожке?

18. Компьютер содержит шину с временем опроса 5 нс. За один цикл опроса он может считать из памяти или записать в память 32-разрядное слово. Компьютер имеет диск Ultra4-SCSI, который использует шину и передает информацию со скоростью 160 Мбайт/с. Центральный процессор обычно вызывает из памяти и выполняет одну 32-разрядную команду каждые 25 нс. На сколько диск замедляет работу процессора?
19. Представьте, что вы пишете программное обеспечение для той части операционной системы, которая отвечает за управление диском. Диск логично представляется как последовательность блоков от 0 на внутренней стороне до какого-либо максимума снаружи. При создании файлов вам приходится выделять память в свободных секторах. Вы можете двигаться от наружного края внутрь или наоборот. Имеет ли значение, какую стратегию выбрать? Поясните свой ответ.
20. Сколько времени занимает считывание диска с 10 000 цилиндрами, каждый из которых содержит 4 дорожки по 2048 секторов? Сначала считываются все сектора дорожки 0, начиная с сектора 0, затем все сектора дорожки 1, начиная с сектора 0 и т. д. Оборот совершается за 10 мс, поиск между соседними цилиндрами занимает 1 мс, а в случае расположения считываемых данных в разных частях диска — до 20 мс. Переход от одной дорожки цилиндра к другой происходит мгновенно.
21. RAID-массив уровня 3 может исправлять одиночные битовые ошибки, используя только один диск четности. Тогда для чего нужен RAID-массив уровня 2? Он ведь тоже может исправлять одиночные ошибки, но использует при этом несколько дисков.
22. Какова точная емкость (в байтах) диска CD-ROM типа 2, содержащего данные на 80 минут (нестандартный объем)? Какова емкость пользовательских данных на диске типа 1?
23. Чтобы записать диск CD-R, лазер должен включаться и выключаться очень быстро. Какова длительность пульсации (включения или выключения) в наносекундах, если компакт-диск типа 1 записывается со скоростью 10x?
24. Чтобы вместить фильм длительностью 133 минуты на односторонний DVD-диск с одним слоем, потребуется сжатие. Вычислите, насколько нужно сжать фильм. Предполагается, что для записи дорожки изображения нужно 3,5 Гбайт, разрешающая способность изображения составляет 720 × 480 пикселов с 24-разрядным цветом и в секунду меняется 30 кадров.
25. Данные с дисков Blu-ray емкостью 25 Гбайт считываются на скорости 4,5 Мбайт/с. Сколько времени требуется на считывание всех данных с такого диска?
26. Производитель говорит, что его цветной графический терминал может воспроизводить 2^{24} различных цветов. Однако аппаратное обеспечение имеет только один байт для каждого пиксела. Каким же образом получается столько цветов?
27. Вы входите в сверхсекретную международную группу ученых, которой только что было поручено исследование Херба — внеземного существа с планеты 10, которое только что прибыло на Землю. Херб сообщил вам следующую информацию о своем зрении. Его поле зрения состоит из 10^8

пикселов. Каждый пиксел представляет собой суперпозицию пяти «цветов» (инфракрасный, красный, зеленый, синий и ультрафиолетовый), каждый из которых может иметь до 32 уровней интенсивности. Временное разрешение поля зрения Херба составляет 10 миллисекунд. Вычислите скорость передачи данных (в Гбайт/с) в глазах Херба.

28. Графический терминал имеет монитор размером 1920 × 1080 пикселов. Изображение на мониторе меняется 75 раз в секунду. Какова продолжительность импульса для одного пиксела?
29. Монохромный лазерный принтер может печатать на одном листе 50 строк по 80 символов определенного шрифта. Символ в среднем занимает пространство 2 × 2 мм, причем тонер занимает 25 % этого пространства, а оставшаяся часть остается белой. Слой тонера составляет 25 микрон в толщину. Картридж с тонером имеет размер 25 × 8 × 2 см. На сколько страниц хватит картриджа?
30. Компания, выпускающая модемы, разработала новый модем с частотной модуляцией, который использует 64 частоты вместо 2. Каждая секунда делится на n равных временны́х отрезков, каждый из которых содержит один из 64 возможных тонов. Сколько битов в секунду может передавать этот модем в случае синхронной передачи?
31. Предположим, что некий пользователь подключился к ADSL со скоростью передачи данных 2 Мбит/с, а его сосед — к линии кабельного Интернета с общей пропускной способностью 12 МГц. Применяется схема модуляции QAM-64. К кабелю подключено n домов, по одному компьютеру в каждом. Часть (f) этих компьютеров в любой отдельно взятый момент времени отключена от сети. При каких условиях скорость передачи данных по кабелю превысит скорость доступа в Интернет по линии ADSL?
32. В цифровой камере с разрешением 3000 × 2000 пикселов для передачи цветов модели RGB на каждый пиксел выделяется 3 байта. Производитель камеры хочет, чтобы изображение в формате JPEG с коэффициентом сжатия 5x можно было записать на карту флэш-памяти за две секунды. При какой скорости передачи данных это требование можно удовлетворить?
33. В профессиональной камере установлен формирователь сигнала изображения на 24 млн пикселов, причем для передачи цвета в каждом из битов выделяется по 6 байт. Сколько изображений можно сохранить на карте флэш-памяти емкостью 8 Гбайт при коэффициенте сжатия 5x? Допустим для упрощения вычислений, что 1 Гбайт равен 2^{30} байт.
34. Оцените, сколько символов (включая пробелы) содержит обычная книга по информатике. Сколько битов нужно для того, чтобы закодировать книгу в коде ASCII с проверкой на четность? Сколько компакт-дисков нужно для хранения 10 000 книг по информатике? Сколько односторонних двухслойных DVD-дисков нужно для хранения такого же количества книг?
35. Напишите процедуру `hamming(ascii, encoded)`, которая преобразует 7 последовательных битов `ascii` в 11-разрядное целое кодированное число `encoded`.
36. Напишите функцию `distance(code, n, k)`, которая на входе получает массив `code` из `n` символов по `k` бит каждый и возвращает минимальное хэммингово расстояние между этими символами.

Глава 3
Цифровой логический уровень

В самом низу иерархической схемы на рис. 1.2 находится цифровой логический уровень, или аппаратное обеспечение компьютера. В этой главе мы рассмотрим различные аспекты цифровой логики, что должно стать основой для изучения более высоких уровней в последующих главах. Предмет изучения находится на границе информатики и электротехники, но материал является самодостаточным, поэтому предварительного ознакомления с аппаратным обеспечением и электротехникой не требуется.

Основные элементы, из которых конструируются цифровые компьютеры, чрезвычайно просты. Сначала мы рассмотрим эти основные элементы, а также специальную двузначную алгебру (булеву алгебру), которая используется при конструировании этих элементов. Затем мы изучим основные схемы, которые можно построить из вентилей в различных комбинациях, в том числе схемы для выполнения арифметических действий. Следующая тема о том, как комбинировать вентили для хранения информации, то есть о том, как построить память. После этого мы перейдем к процессорам и к тому, как процессоры на одной микросхеме обмениваются информацией с памятью и периферийными устройствами. Затем мы рассмотрим несколько реальных примеров из компьютерной отрасли.

Вентили и булева алгебра

Цифровые схемы конструируются из небольшого числа простых элементов путем сочетания этих элементов в различных комбинациях. В следующих подразделах описаны эти основные элементы, показано, как их можно сочетать, а также представлен математический метод анализа их работы.

Вентили

Цифровая схема — это схема, в которой есть только два логических значения. Обычно сигнал от 0 до 1 В представляет одно значение (например, 0), а сигнал от 2 до 5 В — другое значение (например, 1). Напряжение за пределами указанных величин недопустимо. Крошечные электронные устройства, которые называются **вентилями**, позволяют получать различные функции от этих двузначных сигналов. Вентили лежат в основе аппаратного обеспечения, на котором строятся все цифровые компьютеры.

Описание принципов работы вентилей не является темой этой книги, поскольку относится к **уровню физических устройств**, который находится ниже уровня 0. Тем не менее мы очень кратко коснемся основного принципа, который не так уж и сложен. Вся современная цифровая логика основывается на том, что транзистор может работать как очень быстрый двоичный переключатель.

На рис. 3.1, *а* изображен биполярный транзистор, встроенный в простую схему. Транзистор имеет три соединения с внешним миром: **коллектор**, **базу** и **эмиттер**. Если входное напряжение V_{in} ниже определенного критического значения, транзистор выключается и действует как очень большое сопротивление. Это приводит к выходному сигналу V_{out}, близкому к V_{CC} (напряжению, подаваемому извне), — для данного типа транзистора это обычно +5 В. Если V_{in} превышает критическое значение, транзистор включается и действует как проводник, вызывая заземление сигнала V_{out} (по соглашению — это 0 В).

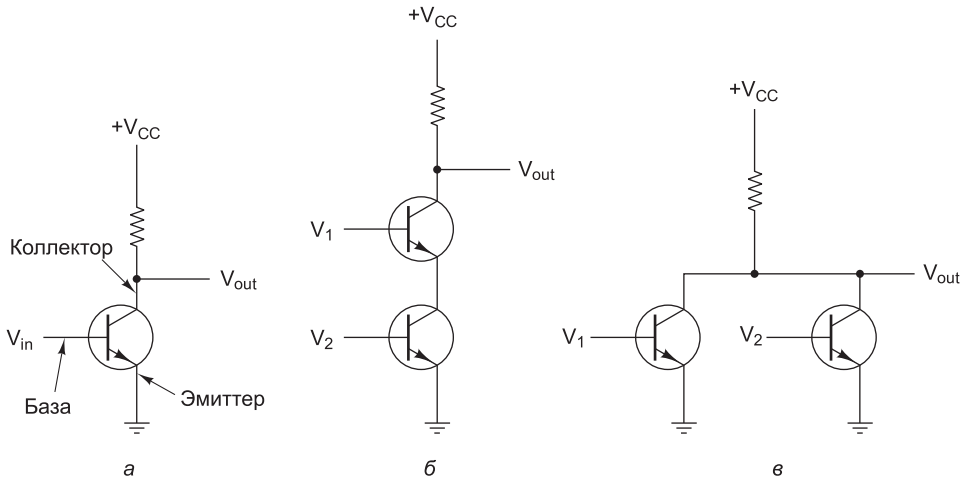


Рис. 3.1. Транзисторный инвертор (*а*); вентиль НЕ И (*б*); вентиль НЕ ИЛИ (*в*)

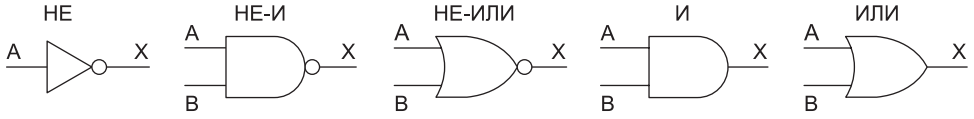
Важно отметить, что если напряжение V_{in} низкое, то V_{out} высокое, и наоборот. Эта схема, таким образом, является инвертором, превращающим логический 0 в логическую 1 и логическую 1 в логический 0. Резистор (ломаная линия) нужен для ограничения тока, проходящего через транзистор, чтобы транзистор не сгорел. На переключение с одного состояния на другое обычно требуется не более наносекунды.

На рис. 3.1, *б* два транзистора соединены последовательно. Если и напряжение V_1, и напряжение V_2 высокое, то оба транзистора становятся проводниками и снижают V_{out}. Если одно из входных напряжений низкое, то соответствующий транзистор выключается, и напряжение на выходе становится высоким. Другими словами, напряжение V_{out} является низким тогда и только тогда, когда и напряжение V_1, и напряжение V_2 высокое.

На рис. 3.1, *в* два транзистора соединены параллельно. Если один из входных сигналов высокий, включается соответствующий транзистор и снижает выходной сигнал. Если оба напряжения на входе низкие, то выходное напряжение становится высоким.

Эти три схемы образуют три простейших вентиля. Они называются вентилями НЕ, НЕ-И и НЕ-ИЛИ соответственно. Вентили НЕ часто называют **инверторами**. Мы будем использовать оба термина. Если мы примем соглашение, что высокое напряжение (V_{CC}) — это логическая 1, а низкое напряжение («земля») — логический 0, то мы сможем выражать значение на выходе как функцию от вход-

ных значений. Значки, которые используются для изображения этих трех типов вентилей, показаны на рис. 3.2, *а*–*в*. Там же показаны режимы работы функции для каждой схемы. На этих рисунках *A* и *B* — входные сигналы, *X* — выходной сигнал. Каждая строка таблицы определяет выходной сигнал для различных комбинаций входных сигналов.



A	X
0	1
1	0

а

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

б

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

в

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

г

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

д

Рис. 3.2. Значки для изображения пяти основных вентилей. Режимы работы функции для каждого вентиля

Если выходной сигнал на рис. 3.1, *б* подать в инвертор, мы получим другую схему, противоположную вентилю НЕ-И, то есть такую схему, у которой выходной сигнал равен 1 тогда и только тогда, когда оба входных сигнала равны 1. Такая схема называется вентилем И; ее схематическое изображение и описание соответствующей функции даны на рис. 3.2, *г*. Точно так же вентиль НЕ-ИЛИ может быть связан с инвертором. Тогда получится схема, у которой выходной сигнал равен 1 в том случае, если хотя бы один из входных сигналов единичный, и равен 0, если оба входных сигнала нулевые. Изображение этой схемы, которая называется вентилем ИЛИ, а также описание соответствующей функции даны на рис. 3.2, *д*. Маленькие кружочки в схемах инвертора, вентиля НЕ-И и вентиля НЕ-ИЛИ называются **инвертирующими выходами**. Они также могут использоваться в другом контексте для указания на инвертированный сигнал.

Пять вентилей, изображенные на рис. 3.2, составляют основу цифрового логического уровня. Из предшествующего обсуждения должно быть ясно, что вентили НЕ-И и НЕ-ИЛИ требуют два транзистора каждый, а вентили И и ИЛИ — три транзистора каждый. По этой причине во многих компьютерах используются вентили НЕ-И и НЕ-ИЛИ, а не И и ИЛИ. (На практике все вентили строятся несколько иначе, но вентили НЕ-И и НЕ-ИЛИ в любом случае проще, чем И и ИЛИ.) Следует упомянуть, что вентили могут иметь более двух входов. В принципе вентиль НЕ-И, например, может иметь произвольное количество входов, но на практике больше восьми обычно не бывает.

Хотя устройство вентилей относится к уровню физических устройств, мы все же упомянем основные линейки производственных технологий, так как они часто упоминаются в литературе. Две основные технологии — **биполярная** и **МОП** (металл, оксид, полупроводник). Среди биполярных технологий можно назвать **ТТЛ** (транзисторно-транзисторная логика), которая служила основой цифровой электроники на протяжении многих лет, и **ЭСЛ** (эмиттерно-связанная

логика), которая используется в тех случаях, когда требуется высокая скорость выполнения операций. В отношении вычислительных схем более распространена технология МОП.

МОП-вентили работают медленнее, чем ТТЛ и ЭСЛ, но потребляют гораздо меньше энергии и занимают гораздо меньше места, поэтому можно компактно расположить большое количество таких вентилей. Вентили МОП имеют несколько разновидностей: *p*-канальный МОП, *n*-канальный МОП и комплиментарный МОП. Хотя МОП-транзисторы конструируются не так, как биполярные транзисторы, они тоже могут функционировать как электронные переключатели. Современные процессоры и память чаще всего производятся с использованием технологии комплиментарных МОП, которая работает при напряжении около +1,5 В. Это все, что мы можем сказать об уровне физических устройств. Читатели, желающие узнать больше об этом уровне, могут обратиться к литературе, приведенной на сайте книги.

Булева алгебра

Чтобы описать схемы, получаемые сочетанием различных вентилей, нужен особый тип алгебры, в которой все переменные и функции могут принимать только два значения: 0 и 1. Такая алгебра называется **булевой**. Она названа в честь английского математика Джорджа Буля (1815–1864). На самом деле, в данном случае мы говорим об особом типе булевой алгебры, а именно — об **алгебре релейных схем**, но термин «булева алгебра» очень часто используется в значении «алгебра релейных схем», поэтому мы не будем их различать.

Как и в обычной алгебре (то есть в той, которую изучают в школе), в булевой алгебре есть свои функции. Булева функция на входе получает одну или несколько переменных и выдает результат, который зависит только от значений этих переменных. Можно определить простую функцию f, сказав, что $f(A) = 1$, если $A = 0$ и $f(A) = 0$, если $A = 1$. Такая функция будет функцией НЕ (см. рис. 3.2, *а*).

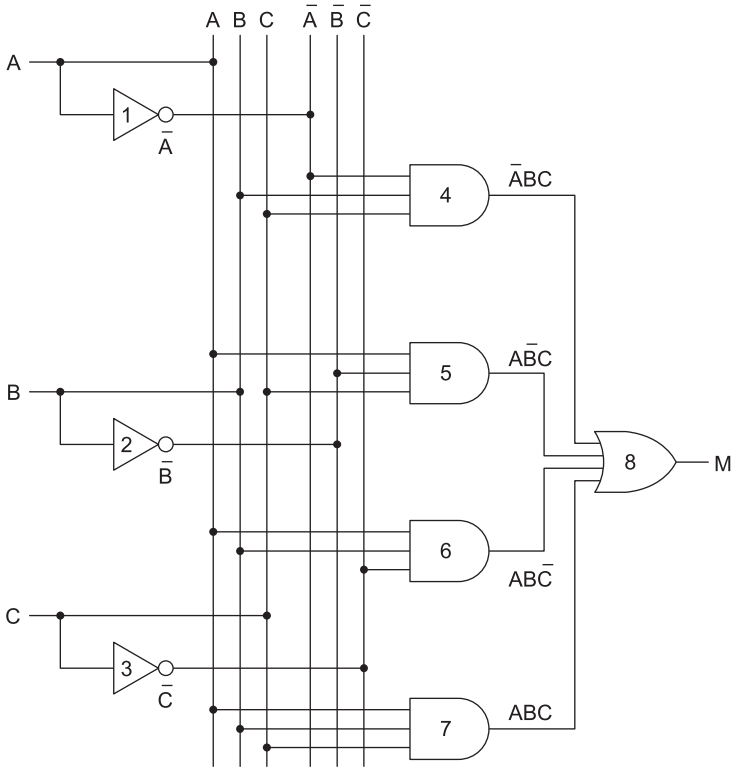
Так как булева функция от n переменных имеет только 2^n возможных комбинаций значений переменных, то такую функцию можно полностью описать в таблице с 2^n строками. В каждой строке будет даваться значение функции для разных комбинаций значений переменных. Такая таблица называется **таблицей истинности**. Все таблицы на рис. 3.2 представляют собой таблицы истинности. Если мы договоримся всегда располагать строки таблицы истинности по порядку номеров, то есть для двух переменных в порядке 00, 01, 10, 11, то функцию можно полностью описать 2^n-разрядным двоичным числом, которое получается, если считывать по вертикали колонку результатов в таблице истинности. Таким образом, НЕ-И — это 1110, НЕ-ИЛИ — 1000, И — 0001 и ИЛИ — 0111. Очевидно, что существуют только 16 булевых функций от двух переменных, которым соответствуют 16 возможных 4-разрядных цепочек. В обычной алгебре, напротив, существует бесконечное число функций от двух переменных, и ни одну из них нельзя описать таблицей значений этой функции для всех допустимых значений входных переменных, поскольку каждая переменная может принимать бесконечное число значений.

На рис. 3.3, *а* показана таблица истинности для булевой функции от трех переменных: $M = f(A, B, C)$. Это функция большинства, которая принимает зна-

чение 0, если большинство переменных равны 0, или 1, если большинство переменных равны 1. Хотя любая булева функция может быть определена с помощью таблицы истинности, с возрастанием количества переменных такой тип записи становится громоздким. Поэтому вместо таблиц истинности часто используется другой вариант записи.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

а



б

Рис. 3.3. Таблица истинности для функции большинства от трех переменных (*а*); схема реализации этой функции (*б*)

Чтобы увидеть этот другой тип записи, отметим, что любую булеву функцию также можно определить указанием комбинаций значений входных переменных, приводящих к единичному значению функции. Для функции на рис. 3.3, *а* существует 4 комбинации переменных, которые дают единичное значение функции. Мы будем рисовать черту над переменной, показывая, что ее значение инвертируется. Отсутствие черты означает, что значение переменной не инвертируется. Кроме того, мы будем использовать знак умножения (точку) для обозначения булевой функции И (этот знак может опускаться) и знак сложения (+) для обозначения булевой функции ИЛИ. Например, $A\overline{B}C$ принимает значение 1, только если $A = 1$, $B = 0$ и $C = 1$. Кроме того, $A\overline{B} + B\overline{C}$ принимает значение 1, только если ($A = 1$ и $B = 0$) или ($B = 1$ и $C = 0$). В таблице на рис. 3.3, *а* функция принимает значение 1 в четырех строках: $\overline{A}BC$, $A\overline{B}C$, $AB\overline{C}$ и ABC. Функция M при-

нимает значение истины (то есть 1), если одно из этих четырех условий истинно. Следовательно, мы можем написать

$$M = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC.$$

Это компактная запись таблицы истинности. Таким образом, функцию от n переменных можно описать «суммой» максимум 2^n «произведений», при этом в каждом «произведении» будет по n множителей. Как мы скоро увидим, такая формулировка особенно важна, поскольку она позволяет реализовать данную функцию с использованием стандартных вентилей.

Важно понимать различие между абстрактной булевой функцией и ее реализацией с помощью электронной схемы. Булева функция состоит из переменных, например A, B и C, а также из операторов И, ИЛИ и НЕ. Булева функция описывается с помощью таблицы истинности или специальной записи, например:

$$F = A\overline{B}C + AB\overline{C}.$$

Булева функция может быть реализована электронной схемой (часто различными способами) с использованием сигналов, которые представляют входные и выходные переменные, и вентилей, например И, ИЛИ и НЕ.

Реализация булевых функций

Как было отмечено ранее, представление булевой функции в виде суммы максимум 2^n произведений подводит нас непосредственно к возможной реализации этой функции. На рис. 3.3, *б* входные сигналы A, B и C показаны с левой стороны, а функция M, полученная на выходе, — с правой. Поскольку входные переменные должны инвертироваться, сигнал проходит через инверторы 1, 2 и 3. Чтобы сделать рисунок понятней, мы нарисовали 6 вертикальных линий, 3 из которых связаны с входными переменными, 3 другие — с их инверсиями. Эти линии обеспечивают передачу входного сигнала к вентилям. Например, вентили 5, 6 и 7 на входе получают сигнал A. В реальной схеме эти вентили, вероятно, будут непосредственно соединены проводом с A без каких-либо промежуточных вертикальных проводов.

Схема содержит четыре вентиля И, по одному для каждого члена в уравнении для M (то есть по одному для каждой строки в таблице истинности с результатом 1). Каждый вентиль И вычисляет одну из указанных строк таблицы истинности. В конце концов, все данные произведения суммируются (имеется в виду операция ИЛИ) для получения конечного результата.

Посмотрите на рис. 3.3, *б*. В этой книге мы будем использовать следующее соглашение: если две линии на рисунке пересекаются, связь подразумевается только в том случае, если на пересечении расположена жирная точка. Например, выход вентиля 3 пересекает все 6 вертикальных линий, но связан он только с линией $\overline{C}$. Отметим, что в других книгах могут использоваться другие соглашения.

Из рис. 3.3 должно быть ясно, как получить схему для любой булевой функции:

1. Составить таблицу истинности для данной функции.
2. Включить в схему инверторы, чтобы иметь возможность инверсии каждого входного сигнала.
3. Нарисовать вентиль И для каждой строки таблицы истинности с результатом 1.

4. Соединить вентили И с соответствующими входными сигналами.
5. Вывести выходы всех вентилей И и направить их на вход вентиля ИЛИ.

Мы показали, как реализовать любую булеву функцию с помощью вентилей НЕ, И и ИЛИ. Однако гораздо удобнее строить схемы с использованием одного типа вентилей. К счастью, схемы, построенные по предыдущему алгоритму, легко преобразуются в форму НЕ-И или НЕ-ИЛИ. Все что нужно для осуществления такого преобразования — это реализовать вентили НЕ, И и ИЛИ с помощью какого-нибудь одного типа вентилей. На рис. 3.4 показано, как это сделать на базе вентилей НЕ-И или НЕ-ИЛИ. (Такое преобразование выполняется элементарно, но существуют и другие варианты.)

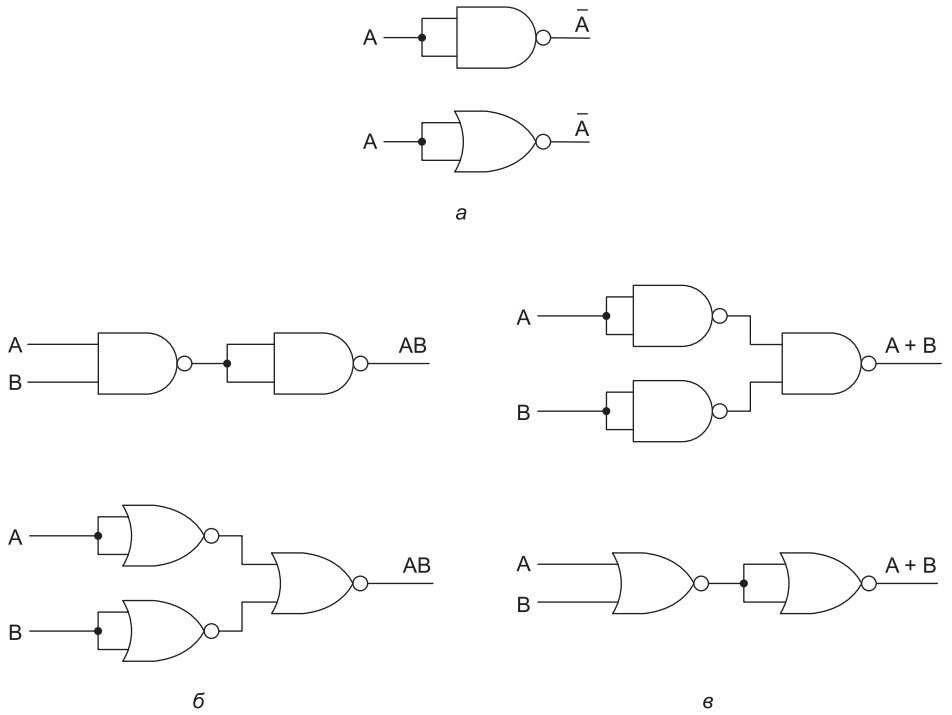


Рис. 3.4. Конструирование вентилей НЕ (*а*), И (*б*) и ИЛИ (*в*) только на базе вентилей НЕ-И или НЕ-ИЛИ

Для того чтобы реализовать булеву функцию только на базе вентилей НЕ-И или НЕ-ИЛИ, можно сначала следовать описанному алгоритму, сконструировав схему с вентилями НЕ, И и ИЛИ. Затем нужно заменить многовходовые вентили эквивалентными схемами на двухвходовых вентилях. Например, $A + B + C + D$ можно поменять на $(A + B) + (C + D)$, используя три двухвходовых вентиля. Затем вентили НЕ, И и ИЛИ заменяются схемами, изображенными на рис. 3.4.

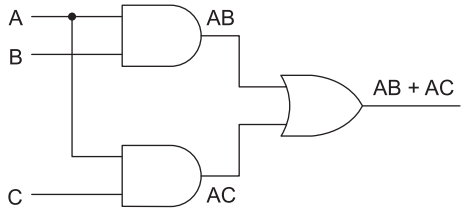
Хотя такая процедура и не приводит к оптимальным схемам с точки зрения минимального числа вентилей, она демонстрирует, что подобное преобразование осуществимо. Вентили НЕ-И и НЕ-ИЛИ считаются **полными**, потому что любая

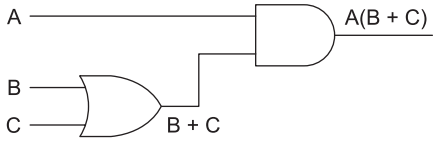
булева функция может быть реализована на их базе. Ни один другой вентиль не обладает таким свойством, вот почему именно эти два типа вентилей предпочтительнее при построении схем.

Эквивалентность схем

Разработчики схем часто стараются сократить число вентилей, чтобы снизить цену, уменьшить занимаемое схемой место, сократить потребление энергии и т. д. Чтобы упростить схему, разработчик должен найти другую схему, которая может вычислять ту же функцию, но при этом требует меньшего количества вентилей (или может работать с более простыми вентилями, например двухвходовыми вместо четырехвходовых). Булева алгебра является ценным инструментом в поиске эквивалентных схем.

В качестве примера использования булевой алгебры рассмотрим схему и таблицу истинности для функции $AB + AC$ (рис. 3.5, *а*). Хотя мы это еще не обсуждали, многие правила обычной алгебры имеют силу и в булевой алгебре. Например, выражение $AB + AC$ по дистрибутивному закону может быть преобразовано в $A(B + C)$. На рис. 3.5, *б* показана схема и таблица истинности для функции $A(B + C)$. Две функции являются эквивалентными тогда и только тогда, если обе функции принимают одно и то же значение для всех возможных переменных. Из таблиц истинности на рис. 3.5 ясно видно, что функция $A(B + C)$ эквивалентна функции $AB + AC$. Несмотря на эту эквивалентность, схема на рис. 3.5, *б* проще, чем схема на рис. 3.5, *а*, поскольку содержит меньше вентилей.





A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

а

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

б

Рис. 3.5. Две эквивалентные функции: AB + AC (*а*); A(B + C) (*б*)

Обычно разработчик исходит из определенной булевой функции, а затем применяет к ней законы булевой алгебры, чтобы найти более простую функцию, эквивалентную исходной. На основе полученной функции можно конструировать схему.

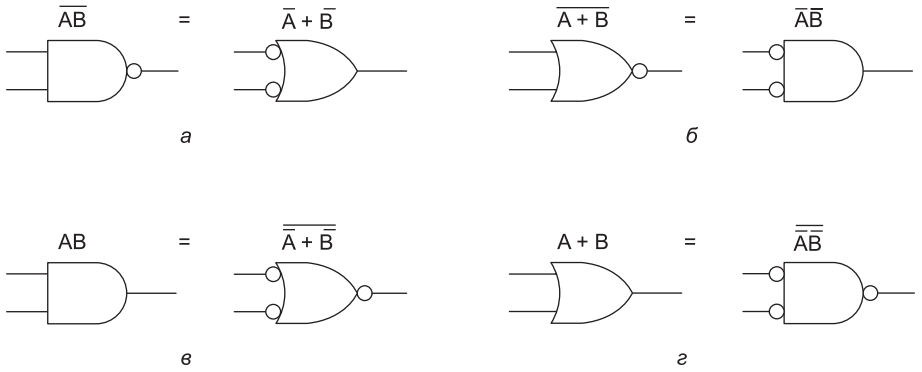
Чтобы использовать данный подход, нужно знать некоторые соотношения (законы) булевой алгебры, которые показаны в табл. 3.1. Интересно отметить, что каждое соотношение имеет две формы. Одну форму можно получить из другой, меняя И на ИЛИ и 0 на 1. Все соотношения можно легко доказать, составив для них таблицы истинности. Почти во всех случаях результаты очевидны, за исключением соотношения Де Моргана, соотношения поглощения и дистрибутивного соотношения. Соотношение Де Моргана может быть расширено на выражения с более чем двумя переменными, например $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$.

Таблица 3.1. Некоторые соотношения булевой алгебры

Соотношение	**И**	**ИЛИ**
Соотношение тождества	$1A = A$	$0 + A = A$
Соотношение нуля	$0A = 0$	$1 + A = 1$
Соотношение идемпотентности	$AA = A$	$A + A = A$
Соотношение инверсии	$A\overline{A} = 0$	$A + \overline{A} = 1$
Соотношение коммутативности	$AB = BA$	$A + B = B + A$
Ассоциативное соотношение	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Дистрибутивное соотношение	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Соотношение поглощения	$A(A + B) = A$	$A + AB = A$
Соотношение Де Моргана	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A} + \overline{B} = \overline{AB}$

Соотношение Де Моргана предполагает альтернативную запись. На рис. 3.7, *а* форма И дается с отрицанием, которое показывается с помощью инвертирующих входов и выходов. Таким образом, вентиль ИЛИ с инвертированными входными сигналами эквивалентен вентилю НЕ-И. Из рис. 3.6, *б*, который иллюстрирует вторую форму соотношения Де Моргана, ясно, что вместо вентиля НЕ-ИЛИ можно нарисовать вентиль И с инвертированными входами. Путем отрицания обеих форм соотношения Де Моргана мы приходим к эквивалентным представлениям вентилей И и ИЛИ (рис. 3.7, *в*, *г*). Аналогичные символические изображения существуют для разных форм соотношения Де Моргана (например, *n*-входовый вентиль НЕ-И становится вентилем ИЛИ с инвертированными входами).

Используя уравнения, указанные на рис. 3.6, и аналогичные уравнения для многовходовых вентилей, можно легко преобразовать сумму произведений в форму только из вентилей НЕ-И или только из вентилей НЕ-ИЛИ. В качестве примера рассмотрим функцию ИСКЛЮЧАЮЩЕЕ ИЛИ (рис. 3.7, *а*). Стандартная схема, выражающая сумму произведений, показана на рис. 3.7, *б*. Чтобы перейти к форме НЕ-И, нужно линии, соединяющие выходы вентилей И с входом вентиля ИЛИ, нарисовать с инвертирующими входами и выходами, как



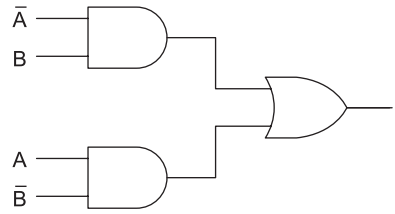
а *б* *в* *г*

Рис. 3.6. Альтернативные представления некоторых вентилей: НЕ-И (*а*); НЕ-ИЛИ (*б*); И (*в*); ИЛИ (*г*)

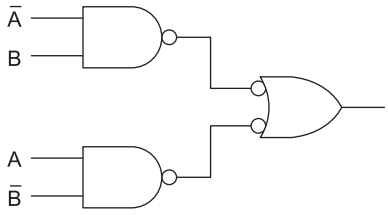
показано на рис. 3.7, *в*. Затем, опираясь на рис. 3.6, *а*, мы приходим к рис. 3.7, *г*. Переменные $\overline{A}$ и $\overline{B}$ можно получить из A и B, используя вентили НЕ-И или НЕ-ИЛИ с объединенными входами. Отметим, что инвертирующие входы (выходы) могут по желанию перемещаться вдоль линии связи, например, от выходов входных вентилей ко входам выходного вентиля.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

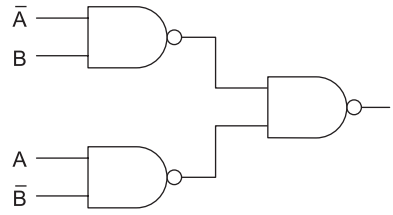
а



б



в



г

Рис. 3.7. Таблица истинности для функции ИСКЛЮЧАЮЩЕЕ ИЛИ (*а*). Три схемы для вычисления этой функции (*б*–*г*)

Остается отметить, что один и тот же физический вентиль может вычислять разные функции в зависимости от используемых соглашений. На рис. 3.8, *а* мы показали выходные сигналы вентиля F для различных комбинаций входных сигналов. И входные, и выходные сигналы даны в вольтах. Если мы примем

соглашение, что 0 В — это логический ноль, а 3,3 В или 5 В — логическая единица, мы получим таблицу истинности, показанную на рис. 3.8, *б*, то есть функцию И. Такое соглашение называется **позитивной логикой**. Однако если мы примем **негативную логику**, то есть условимся, что 0 В — это логическая единица, а 3,3 В или 5 В — логический ноль, то мы получим таблицу истинности, показанную на рис. 3.8, *в*, то есть функцию ИЛИ.

A	B	F
0 В	0 В	0 В
0 В	5 В	0 В
5 В	0 В	0 В
5 В	5 В	5 В

а

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

б

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

в

Рис. 3.8. Электрические характеристики устройства (*а*); позитивная логика (*б*); негативная логика (*в*)

Таким образом, все зависит от того, какое соглашение выбрано для отображения напряжений в логических величинах. В этой книге мы в основном ограничимся позитивной логикой, так что термины «логическое значение 1», «истина» и «высокий уровень» считаются синонимами, как и термины «логическое значение 0», «ложь» и «низкий уровень».

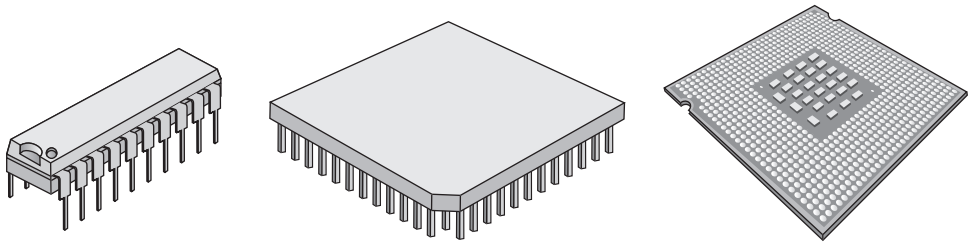
Основные цифровые логические схемы

Теперь мы знаем, как реализовать таблицы истинности и строить простейшие схемы из отдельных вентилей. На практике цифровые логические схемы очень редко строятся вентиль за вентилем, хотя когда-то такой подход был распространен. Сейчас стандартные «строительные» блоки представляют собой модули, объединяющие несколько вентилей. В следующих подразделах мы рассмотрим эти стандартные блоки более подробно и увидим, как они используются и как построить их из отдельных вентилей.

Интегральные схемы

Вентили производятся и продаются не по отдельности, а в модулях, которые называются **интегральными схемами (ИС)**, или **микросхемами**. Интегральная схема представляет собой квадратный кусок кремния, размер которого зависит от количества вентилей, необходимых для реализации компонентов. Размеры маленьких интегральных схем обычно составляют около 2 × 2 мм, большие микросхемы могут иметь размеры около 18 × 18 мм. Микросхемы обычно помещаются в прямоугольные пластиковые или керамические корпуса значительно большего размера, если для обмена данными с внешним миром микросхеме требуется много выводов. Каждый вывод соединяется с входом или выходом какого-нибудь вентиля, с источником питания или с «землей».

На рис. 3.9 изображены некоторые распространенные варианты корпусов микросхем. Для небольших микросхем (например, используемых в микроконтроллерах или схемах памяти), используются **корпуса с двусторонним расположением выводов** (Dual Inline Package, **DIP**) — корпуса с двумя рядами выводов, вставляемых в соответствующие гнезда на материнской плате. Обычно корпуса DIP имеют 14, 16, 18, 20, 22, 24, 28, 40, 64 или 68 выводов. Для больших микросхем часто используются корпуса, у которых выводы расположены со всех четырех сторон или снизу. Два распространенных корпуса для микросхем большего размера — **PGA** (Pin Grid Array) и **LGA** (Land Grid Array). У корпусов PGA выводы располагаются на нижней стороне корпуса и входят в соответствующие гнезда на материнской плате. Сокеты PGA часто используют механизм нулевого усилия вставки; другими словами, PGA вставляется в гнездо без усилий, после чего нажатие рычага прикладывает давление ко всем выводам PGA, а микросхема прочно удерживается в гнезде. У микросхем LGA на нижней поверхности корпуса располагаются контактные площадки, а у гнезда LGA имеется крышка, которая прижимает микросхему к плате и обеспечивает контакт площадок LGA с площадками гнезда.



а *б* *в*

Рис. 3.9. Основные типы корпусов интегральных микросхем: DIP (*а*), PGA (*б*) и LGA (*в*)

Корпуса микросхем часто имеют симметричную форму, поэтому при их установке вечно возникают проблемы с выбором ориентации. Корпуса DIP обычно имеют с одной стороны выемку, которая должна соответствовать метке на гнезде DIP. У корпусов PGA обычно отсутствует один вывод, так что попытка неправильно вставить PGA в гнездо обречена на неудачу. Так как у корпусов LGA нет выводов, правильность установки обеспечивается размещением выемок на одной или двух сторонах LGA. Если выемки не соответствуют ключам в гнезде LGA, микросхема не войдет в гнездо.

Для удобства мы считаем, что выходной сигнал вентиля изменяется, как только изменяется сигнал на его входе. На самом деле существует определенная **задержка вентиля**, которая включает в себя время прохождения сигнала через микросхему и время переключения. Задержка обычно составляет от сотен пикосекунд до нескольких наносекунд.

В настоящее время стало возможным помещать до 1 млрд транзисторов на одну микросхему[1]. Так как любая схема может быть сконструирована из вентилей НЕ-И, может создаться впечатление, что производитель способен изготовить

[1] Не стоит забывать закон Мура. Ядро процессора Pentium IV содержит уже 42 млн транзисторов, и, очевидно, это не предел. — *Примеч. науч. ред.*

микросхему, содержащую 500 млн вентилей НЕ-И. К несчастью, для создания такой микросхемы потребуется 1 500 000 002 выводов. Поскольку стандартное расстояние между выводами составляет 1 мм, микросхема LGA будет иметь в длину более 38 м, что, вероятно, отрицательно скажется на ее рыночных свойствах. Поэтому чтобы использовать преимущество данной технологии, нужно разработать такие схемы, в которых количество вентилей значительно превышает количество выводов. В следующих подразделах мы рассмотрим простые микросхемы, в которых для вычисления той или иной функции несколько вентилей соединяются определенным образом между собой, что позволяет уменьшить число внешних выводов.

Комбинаторные схемы

Многие применения цифровой логики требуют наличия схем с несколькими входами и несколькими выходами, в которых выходные сигналы определяются текущими входными сигналами. Такая схема называется **комбинаторной**. Не все схемы обладают таким свойством. Например, схема, содержащая элементы памяти, может генерировать выходные сигналы, которые зависят от значений, хранящихся в памяти. Микросхема, которая реализует таблицу истинности (например, приведенную на рис. 3.3, *а*), является типичным примером комбинаторной схемы. В этом разделе мы рассмотрим наиболее часто используемые комбинаторные схемы.

Мультиплексоры

На цифровом логическом уровне **мультиплексор** представляет собой схему с 2^n входами, одним выходом и n линиями управления, которые позволяют выбрать один из входов. Выбранный вход соединяется с выходом. На рис. 3.10 изображена схема восьмивходового мультиплексора. Три линии управления A, B и C кодируют 3-разрядное число, которое указывает, какая из восьми входных линий должна соединяться с вентилем ИЛИ и, следовательно, с выходом. Вне зависимости от того, какое значение окажется на линиях управления, семь вентилей И всегда будут выдавать на выходе 0, а оставшийся может выдавать 0 или 1 в зависимости от значения выбранной линии входа. Каждый вентиль И запускается определенной комбинацией сигналов на линиях управления. Схема мультиплексора показана на рис. 3.10.

Используя мультиплексор, мы можем реализовать функцию большинства (см. рис. 3.3, *а*), как показано на рис. 3.11, *б*. Для каждой комбинации A, B и C выбирается одна из входных линий. Каждый вход соединяется либо с сигналом V_{CC} (логическая 1), либо с землей (логический 0). Алгоритм соединения входов очень прост: входной сигнал D_i такой же, как значение в строке i таблицы истинности. На рис. 3.3, *а* в строках 0, 1, 2 и 4 значение функции равно 0, поэтому соответствующие входы заземляются; в оставшихся строках значение функции равно 1, поэтому соответствующие входы соединяются с логической единицей. Таким способом на базе микросхемы на рис. 3.11, *а* можно реализовать любую таблицу истинности с тремя переменными.

Мы уже видели, как мультиплексор может использоваться для выбора одного из нескольких входов и как он позволяет строить таблицу истинности. Его также

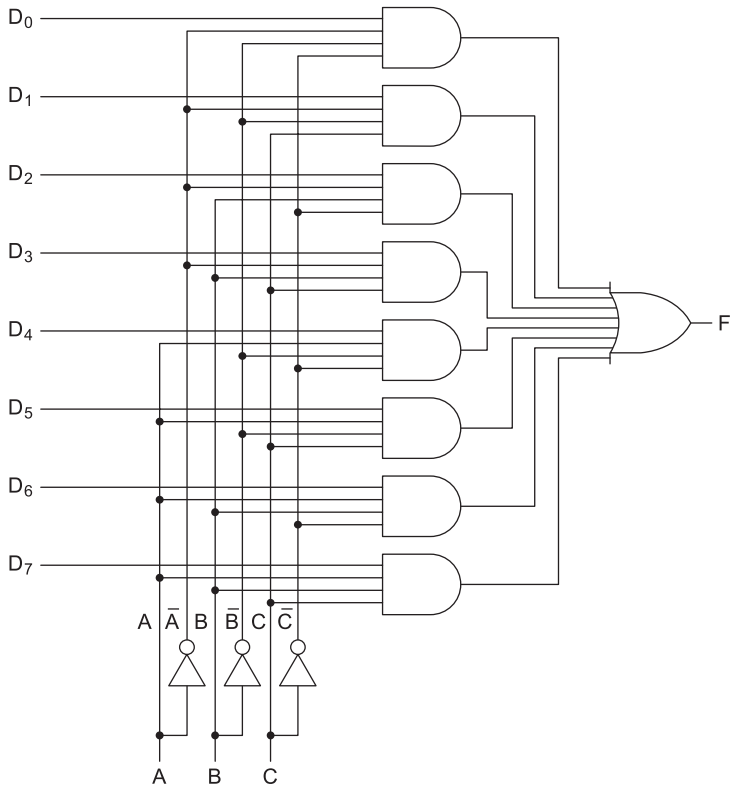
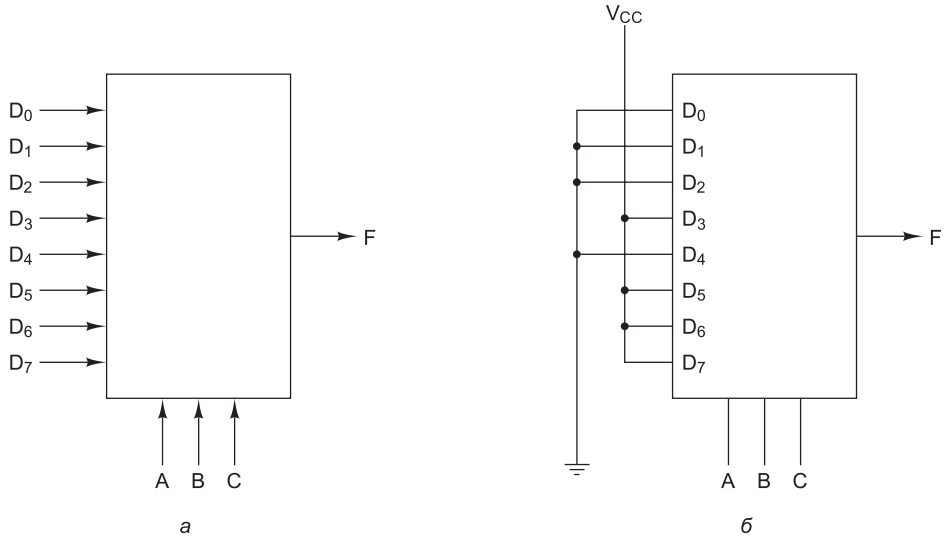


Рис. 3.10. Схема восьмивходового мультиплексора



а *б*

Рис. 3.11. Восьмивходовой мультиплексор (*а*); тот же мультиплексор, смонтированный для вычисления функции большинства (*б*)

можно использовать в качестве преобразователя параллельного кода в последовательный. Если подать 8 бит данных на входные линии, а затем поочередно переключать линии управления, чтобы получить значения от 000 до 111 (это двоичные числа), то 8 бит поступят на выходную линию последовательно. Обычно такое преобразование осуществляется при вводе информации с клавиатуры, поскольку каждое нажатие клавиши определяет 7- или 8-разрядное число, которое должно передаваться последовательно по телефонной линии.

Противоположностью мультиплексора является **демультиплексор**, который соединяет единственный входной сигнал с одним из 2^n выходов в зависимости от значений сигналов в n линиях управления. Если бинарное значение линий управления равно k, то выбирается выход k.

Декодеры

В качестве второго примера рассмотрим схему, которая получает на входе n-разрядное число и использует его для того, чтобы выбрать (то есть установить в значение 1) одну из 2^n выходных линий. Такая схема называется **декодером**. Пример декодера для $n = 3$ показан на рис. 3.12.

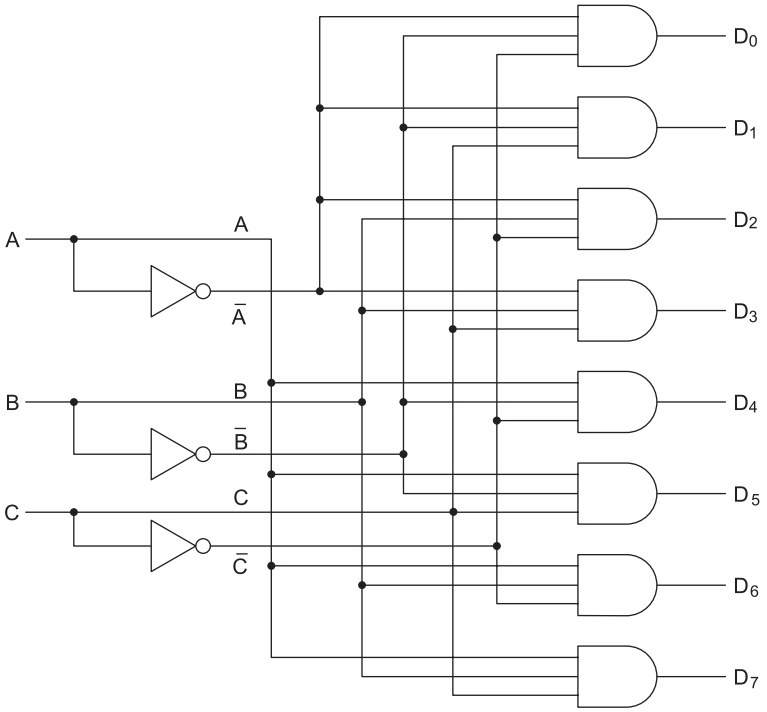


Рис. 3.12. Схема декодера, содержащего 3 входа и 8 выходов

Чтобы понять, зачем нужен декодер, представим себе память, состоящую из восьми микросхем, каждая из которых содержит 256 Мбайт. Микросхема 0 имеет адреса от 0 до 256 Мбайт, микросхема 1 — адреса от 256 до 512 Мбайт и т. д. Три старших двоичных разряда адреса используются для выбора одной из восьми микросхем. На рис. 3.12 эти три бита — три входа *A*, *B* и *C*. В зависимости от

входных сигналов ровно одна из восьми выходных линий (D_0, ..., D_7) принимает значение 1; остальные линии принимают значение 0. Каждая выходная линия активизирует одну из восьми микросхем памяти. Поскольку только одна линия принимает значение 1, активизируется только одна микросхема.

Принцип работы схемы, изображенной на рис. 3.12, не сложен. Каждый вентиль И имеет три входа, из которых первый — это A или $\overline{A}$, второй — B или $\overline{B}$, третий — C или $\overline{C}$. Каждый вентиль запускается различной комбинацией входов: D_0 — сочетанием $\overline{A}\ \overline{B}\ \overline{C}$, D_1 — сочетанием $\overline{A}\ \overline{B}\ C$ и т. д.

Компараторы

Еще одна полезная схема — **компаратор**. Компаратор сравнивает два слова, которые поступают на вход. Компаратор, изображенный на рис. 3.13, принимает два входных сигнала A и B по 4 бита каждый и выдает 1, если они равны, и 0, если они не равны. Схема основывается на вентиле ИСКЛЮЧАЮЩЕЕ ИЛИ, который выдает 0, если сигналы на входе равны, и 1, если сигналы на входе не равны. Если все четыре входных слова равны, все четыре вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ должны выдавать 0. Эти четыре сигнала затем поступают в вентиль ИЛИ. Если в результате получается 0, значит, слова, поступившие на вход, равны; в противном случае они не равны. В нашем примере мы использовали вентиль ИЛИ в качестве конечного, чтобы поменять значение полученного результата: 1 означает равенство, 0 — неравенство.

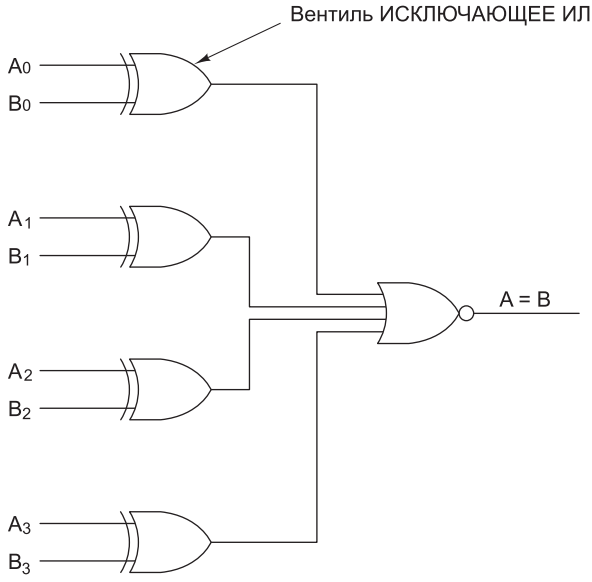


Рис. 3.13. Простой 4-разрядный компаратор

Арифметические схемы

Перейдем от схем общего назначения к комбинаторным схемам, которые используются для выполнения арифметических операций. Напомним, что для комбинационных схем состояние выходов является функцией состояния входных сигна-

лов, но схемы, используемые для выполнения арифметических операций, этим свойством не обладают. Мы начнем с простой 8-разрядной схемы сдвига, затем рассмотрим структуру сумматоров и, наконец, изучим арифметико-логические устройства, которые играют существенную роль в любом компьютере.

Схемы сдвига

Первой арифметической схемой, которую мы рассмотрим, будет схема сдвига с 8 входами и 8 выходами (рис. 3.14). Восемь входных битов подаются на линии D_0, ..., D_7. Выходные данные, которые представляют собой входные данные, сдвинутые на один бит, поступают на линии S_0, ..., S_7. Линия управления C определяет направление сдвига: 0 — влево, 1 — вправо. При сдвиге влево в бит 7 вставляется 0. Аналогичным образом при сдвиге вправо в бит 0 вставляется значение 1.

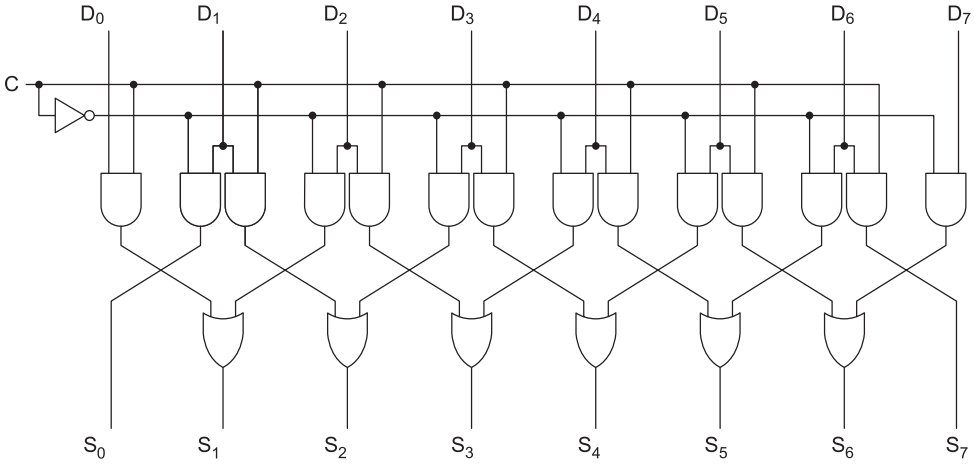


Рис. 3.14. Схема сдвига

Чтобы понять, как работает такая схема, рассмотрим пары вентилей И (кроме крайних). Если C = 1, правый член каждой пары включается, пропуская через себя соответствующий бит. Так как правый вентиль И соединен с входом вентиля ИЛИ, который расположен справа от этого вентиля И, происходит сдвиг вправо. Если C = 0, включается левый вентиль И из пары, и тогда происходит сдвиг влево.

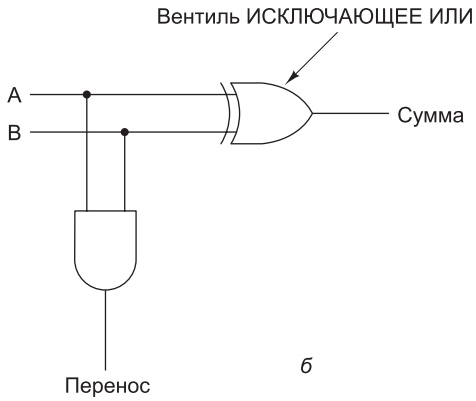
Сумматоры

Компьютер, который не умеет складывать целые числа, практически немыслим. Следовательно, схема выполнения операций сложения является существенной частью любого процессора. Таблица истинности для сложения одноразрядных целых чисел показана на рис. 3.15, *а*. Здесь имеется два результата: сумма входных переменных A и B и перенос на следующую (левую) позицию. Схема для вычисления бита суммы и бита переноса показана на рис. 3.15, *б*. Такая схема обычно называется **полусумматором**.

Полусумматор подходит для сложения битов нижних разрядов двух многобитовых слов. Однако он не годится для сложения битов в середине слова, потому что не может осуществлять перенос в эту позицию. Поэтому необходим **полный**

A	B	Сумма	Перенос
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

а



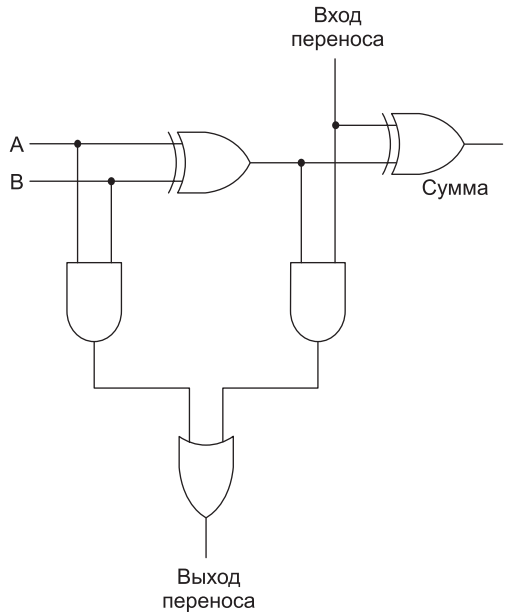
б

Рис. 3.15. Таблица истинности для сложения одноразрядных чисел (*а*); схема полусумматора (*б*)

сумматор (рис. 3.16). Из схемы должно быть ясно, что полный сумматор состоит из двух полусумматоров. Сумма равна 1, если нечетное число переменных *A*, *B* и *вход переноса* принимает значение 1 (то есть если единице равна или одна из переменных или все три). *Выход переноса* принимает значение 1, если либо *A* и *B* одновременно равны 1 (левый вход в вентиль ИЛИ), либо один из них равен 1 и *вход переноса* также равен 1. Два полусумматора порождают и биты суммы, и биты переноса.

A	B	Вход переноса	Сумма	Выход переноса
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

а



б

Рис. 3.16. Таблица истинности для полного сумматора (*а*); схема для полного сумматора (*б*)

Чтобы построить сумматор, например, для двух 16-разрядных слов, нужно 16 раз продублировать схему, изображенную на рис. 3.16, *б*. Перенос производится в левый соседний бит. Перенос в самый правый бит соединен с 0. Такой сумматор называется **сумматором со сквозным переносом**. Прибавление 1 к числу 111...111 не осуществится до тех пор, пока перенос не пройдет весь путь от самого правого бита к самому левому. Существуют более быстрые сумматоры, работающие без подобной задержки. Естественно, предпочтение обычно отдается им.

Рассмотрим пример более быстрого сумматора. Разобьем 32-разрядный сумматор на две половины: нижнюю 16-разрядную и верхнюю 16-разрядную. Когда начинается сложение, верхний сумматор еще не может приступить к работе, поскольку не знает значение переноса, а узнать его он не сможет, пока не совершится 16 суммирований в нижнем сумматоре.

Однако можно внести в схему одно изменение. Вместо одного верхнего сумматора можно получить два верхних сумматора, продублировав соответствующую часть аппаратуры. Тогда схема будет состоять из трех 16-разрядных сумматоров: одного нижнего и двух верхних U_0 и U_1, работающих параллельно. В качестве переноса в сумматор U_0 поступает 0, в сумматор U_1 — 1. Оба верхних сумматора начинают работать одновременно с нижним сумматором, но только один из результатов суммирования в двух верхних сумматорах будет правильным. После сложения 16 нижних разрядов становится известно значение переноса в верхний сумматор, и тогда можно определить правильный ответ. При подобном подходе время сложения сокращается в два раза. Такой сумматор называется **сумматором с выбором переноса**. Можно еще раз разбить каждый 16-разрядный сумматор на два 8-разрядных и т. д.

Арифметико-логические устройства

Большинство компьютеров содержат одну схему для выполнения над двумя машинными словами операций И, ИЛИ и сложения. Обычно эта схема для n-разрядных слов состоит из n идентичных схем — по одной для каждой битовой позиции. На рис. 3.17 изображена такая схема, которая называется **арифметико-логическим устройством (АЛУ)**. Это устройство может вычислять одну из 4-х следующих функций: A И B, A ИЛИ B, $\overline{B}$ или $A + B$. Выбор функции зависит от того, какие сигналы поступают на линии F_0 и F_1: 00, 01, 10 или 11 (в двоичной системе счисления). Отметим, что здесь $A + B$ означает арифметическую сумму A и B, а не логическую операцию ИЛИ.

В левом нижнем углу схемы находится двухразрядный декодер, который генерирует сигналы включения для четырех операций. Выбор операции определяется сигналами управления F_0 и F_1. В зависимости от значений F_0 и F_1 выбирается одна из четырех линий разрешения, и тогда выходной сигнал выбранной функции проходит через последний вентиль ИЛИ.

В верхнем левом углу схемы находится логическое устройство для вычисления функций A И B, A ИЛИ B и $\overline{B}$, но только один из этих результатов проходит через последний вентиль ИЛИ в зависимости от того, какую из линий разрешения выбрал декодер. Так как ровно один из выходных сигналов декодера может быть равен 1, то и запускаться будет ровно один из четырех вентилей И. Остальные три вентиля будут выдавать 0 независимо от значений A и B.

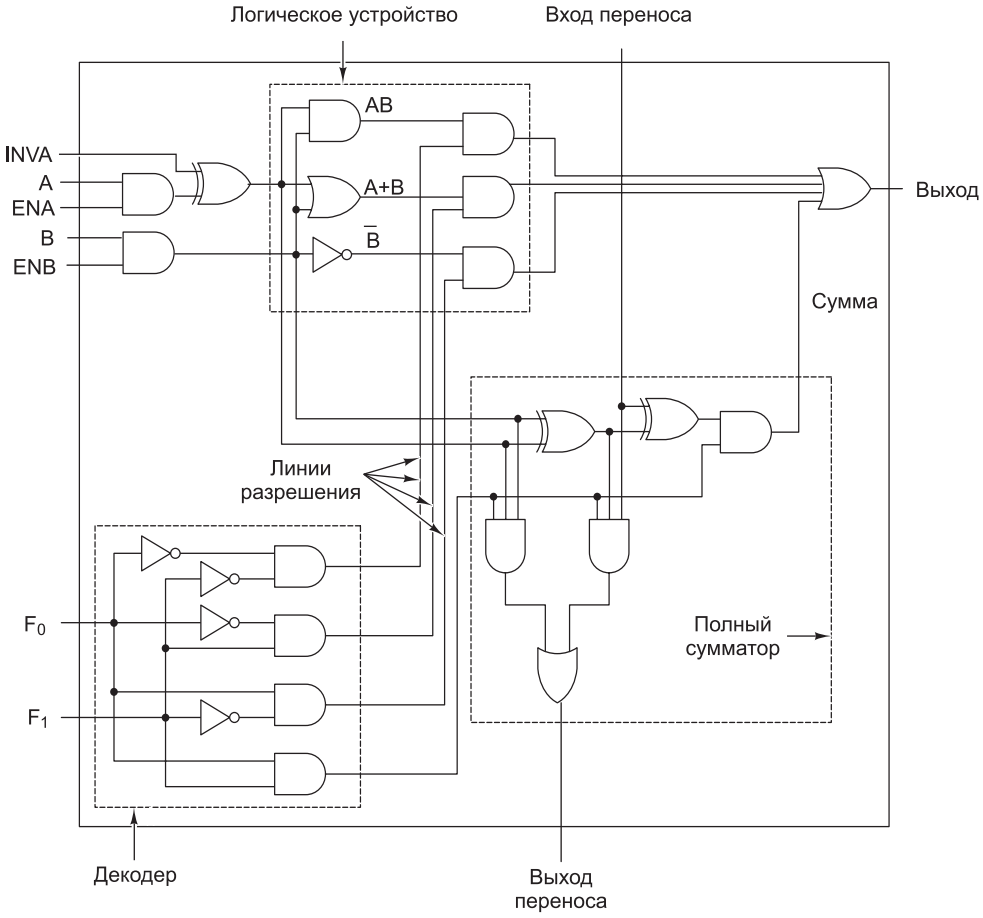


Рис. 3.17. Одноразрядное АЛУ

АЛУ может выполнять не только логические и арифметические операции над переменными *A* и *B*, но и делать их равными нулю, отрицая ENA (сигнал разрешения *A*) или ENB (сигнал разрешения *B*). Можно также получить $\overline{A}$, установив сигнал INVA (инверсия *A*). Зачем нужны сигналы ENA, ENB и INVA, мы узнаем в главе 4. При нормальных условиях и ENA, и ENB равны 1, чтобы разрешить поступление обоих входных сигналов, а сигнал INVA равен 0. В этом случае *A* и *B* просто поступают в логическое устройство без изменений.

В нижнем правом углу находится полный сумматор для подсчета суммы *A* и *B*, а также для осуществления переносов. Переносы необходимы, поскольку несколько таких схем могут быть соединены для выполнения операций над целыми словами. Одноразрядные схемы, подобные показанной на рис. 3.17, называются разрядными микропроцессорными секциями. Они позволяют разработчику строить АЛУ любой разрядности. На рис. 3.18 показана схема 8-разрядного АЛУ, составленного из восьми **одноразрядных секций**. Сигнал INC (увеличение на единицу) нужен только для операций сложения. Он дает возможность вычислять такие суммы, как $A + 1$ и $A + B + 1$.

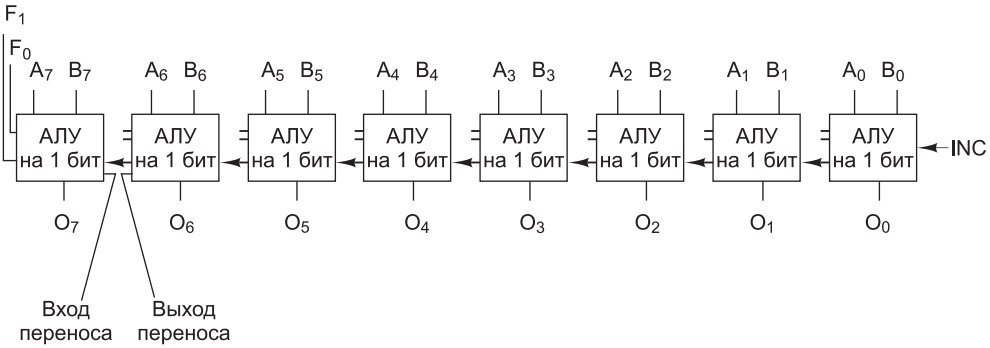


Рис. 3.18. Восемь одноразрядных секций, соединенных в 8-разрядное АЛУ. Для упрощения схемы сигналы разрешения и инверсии не показаны

Несколько лет назад одноразрядная секция была реальной микросхемой, которую можно было купить. В наши дни она скорее представляет собой библиотеку, которую проектировщик микросхемы дублирует нужное количество раз в программе автоматизированного проектирования; программа выдает выходной файл, управляющий оборудованием по производству микросхем. Впрочем, общая идея осталась неизменной.

Тактовые генераторы

Во многих цифровых схемах все зависит от порядка выполнения операций. Иногда одна операция должна предшествовать другой, иногда две операции должны происходить одновременно. Для контроля временны́х параметров в цифровые схемы встраиваются тактовые генераторы, позволяющие обеспечить синхронизацию. **Тактовый генератор** — это схема, которая вызывает серию импульсов. Все импульсы одинаковы по длительности. Интервалы между последовательными импульсами также одинаковы. Временной интервал между началом одного импульса и началом следующего называется **временем такта**. Частота импульсов обычно составляет от 100 МГц до 4 ГГц, что соответствует времени такта от 10 до 250 пс. Частота тактового генератора обычно контролируется высокоточным кварцевым генератором.

В компьютере за один такт может произойти множество событий. Если они должны осуществляться в определенном порядке, то такт следует разделить на подтакты. Чтобы достичь лучшего разрешения, чем у основного тактового генератора, нужно сделать ответвление от задающей линии тактового генератора и вставить схему с определенным временем задержки. Так порождается вторичный сигнал тактового генератора, сдвинутый по фазе относительно первичного (рис. 3.19, *а*). Временна́я диаграмма, показанная на рис. 3.19, *б*, предлагает четыре точки начала отсчета времени для дискретных событий:

1. Фронт C_1.
2. Спад C_1.
3. Фронт C_2.
4. Спад C_2.

Связав различные события с разными перепадами (фронтами и спадами), можно достичь требуемой последовательности выполнения действий. Если в пределах одного такта нужно более четырех точек начала отсчета, можно сделать еще несколько ответвлений от задающей линии с различным временем задержки.

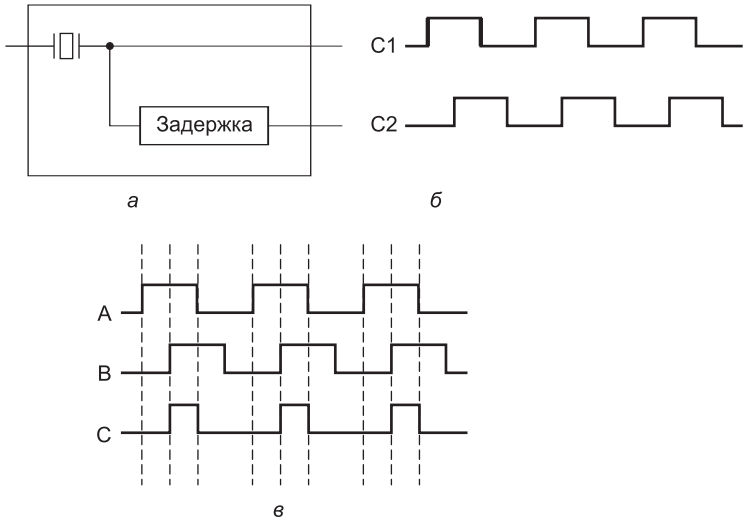


Рис. 3.19. Тактовый генератор (*а*); временна́я диаграмма тактового генератора (*б*); порождение асинхронных тактовых импульсов (*в*)

В некоторых схемах важны временны́е интервалы, а не дискретные моменты времени. Например, некоторое событие может происходить не на фронте импульса, а в любое время, когда уровень импульса C_1 высокий. Другое событие может происходить только в том случае, когда уровень импульса C_2 высокий. Если необходимо более двух интервалов, нужно предоставить больше линий передачи синхронизирующих импульсов или сделать так, чтобы состояния с высоким уровнем импульса у двух тактовых генераторов частично пересекались во времени. В последнем случае можно выделить 4 отдельных интервала: $\overline{C}_1$ И $\overline{C}_2$, $\overline{C}_1$ И C_2, C_1 И $\overline{C}_2$ и C_1 И C_2.

Тактовые генераторы могут быть синхронными. В этом случае время существования импульса с высоким уровнем равно времени существования импульса с низким уровнем (см. рис. 3.19, *б*). Чтобы получить асинхронную серию импульсов (см. сигнал *C* на рис. 3.19, *в*), нужно сдвинуть сигнал задающего генератора, используя цепь задержки. Затем полученный сигнал соединяется с изначальным сигналом с помощью логической функции И.

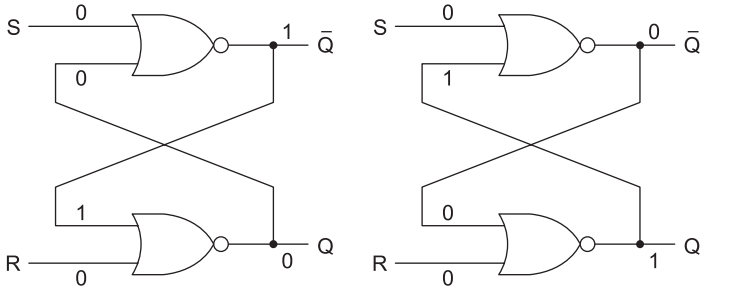
Память

Память является необходимым компонентом любого компьютера. Без памяти не было бы компьютеров, по крайней мере таких, какие есть сейчас. Память используется для хранения как команд, так и данных. В следующих подразделах мы рас-

смотрим основные компоненты памяти, начиная с уровня вентилей. Мы увидим, как они работают, как из них можно построить память большой емкости.

Защелки

Чтобы создать один бит памяти, нужна схема, которая каким-то образом «запоминает» предыдущие входные значения. Такую схему можно сконструировать из двух вентилей НЕ-ИЛИ, как показано на рис. 3.20, *а*. Аналогичные схемы можно построить из вентилей НЕ-И. Мы не будем упоминать эти схемы в дальнейшем, поскольку они по существу идентичны схемам с вентилями НЕ-ИЛИ.



A	B	НЕ-ИЛИ
0	0	1
0	1	0
1	0	0
1	1	0

в

Рис. 3.20. Защелка НЕ-ИЛИ в состоянии 0 (*а*); защелка НЕ-ИЛИ в состоянии 1 (*б*); таблица истинности для функции НЕ-ИЛИ (*в*)

Схема, изображенная на рис. 3.20, *а*, называется **SR-защелкой**. У нее есть два входа: S (Setting — установка) и R (Resetting — сброс). У нее также есть два комплиментарных выхода: Q и $\overline{Q}$. В отличие от комбинаторной схемы, выходные сигналы защелки не определяются текущими входными сигналами.

Чтобы понять, как работает защелка, предположим, что $S = 0$ и $R = 0$ (вообще сигнал на этих входах равен 0 большую часть времени). Предположим также, что $Q = 0$. Так как Q возвращается в верхний вентиль НЕ-ИЛИ и оба входа этого вентиля равны 0, то его выход, $\overline{Q}$, равен 1. Единица возвращается в нижний вентиль, у которого в итоге один вход равен 0, другой — 1, а на выходе получается $Q = 0$. Такое состояние по крайней мере логически последовательно (см. рис. 3.20, *а*).

А теперь давайте представим, что $Q = 1$, а R и S все еще равны 0. Верхний вентиль имеет входы 0 и 1 и выход Q (то есть 0), который возвращается в нижний вентиль. Такое состояние, изображенное на рис. 3.20, *б*, также логически последовательно. Положение, когда оба выхода равны 0, не является логически последовательным, поскольку в этом случае оба вентиля имели бы на входе два нуля, что привело бы к единице на выходе, а не к нулю. Точно так же невозможно иметь оба выхода равные 1, поскольку это привело бы к входным сигналам 0 и 1, что вызывает на выходе 0, а не 1. Наш вывод прост: при $R = S = 0$ защелка имеет два устойчивых состояния, которые мы будем называть 0 и 1 в зависимости от Q.

А сейчас давайте рассмотрим действие входных сигналов на состояние защелки. Предположим, что S принимает значение 1, в то время как $Q = 0$. Тогда входные сигналы верхнего вентиля равны 1 и 0, что ведет к выходному сигналу

$\overline{Q} = 0$. Это изменение делает оба входа в нижний вентиль равными 0, и, следовательно, выходной сигнал равняется 1. Таким образом, установка S в значение 1 переключает состояние с 0 на 1. Установка R в значение 1, когда защелка находится в состоянии 0, не вызывает изменений, поскольку выход нижнего вентиля НЕ-ИЛИ равен 0 как для входов 10, так и для входов 11.

Используя подобные рассуждения, легко увидеть, что установка S в значение 1 при состоянии защелки 1 (то есть при $Q = 1$) не вызывает изменений, но установка R в значение 1 приводит к изменению состояния защелки. Таким образом, если S принимает значение 1, то Q равняется 1 независимо от предыдущего состояния защелки. Сходным образом переход R в значение 1 вызывает $Q = 0$. Схема «запоминает», какой сигнал был последним: S или R. Используя это свойство, мы можем строить компьютерную память.

Синхронные SR-защелки

Часто бывает удобно, чтобы состояние защелки могло изменяться только в определенные моменты. Чтобы достичь этой цели, немного изменим основную схему и получим **синхронную SR-защелку** (рис 3.21).

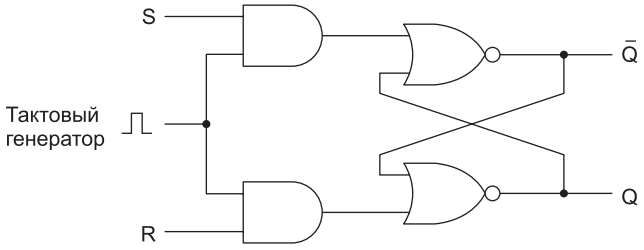


Рис. 3.21. Синхронная SR-защелка

Эта схема имеет дополнительный синхронизирующий вход, который по большей части равен 0. Если этот вход равен 0, то оба выхода вентилей И равны 0, и независимо от значений S и R защелка не меняет свое состояние. Когда значение синхронизирующего входа равно 1, действие вентилей И прекращается, и состояние защелки становится зависимым от S и R. Для обозначения факта появления единицы на синхронизирующем входе часто используются термины **включение** и **стробирование**.

До сих пор мы не выясняли, что происходит, когда $S = R = 1$. И по понятным причинам: когда и R, и S в конце концов возвращаются к 0, схема становится недетерминированной. Единственное приемлемое состояние при $S = R = 1$ — это $Q = \overline{Q} = 0$, но как только оба входа возвращаются к 0, защелка должна перейти в одно из двух устойчивых состояний. Если один из входов принимает значение 0 раньше, чем другой, оставшийся в состоянии 1 «побеждает», потому что именно единичный вход управляет состоянием защелки. Если оба входа переходят к 0 одновременно (что очень маловероятно), защелка выбирает одно из своих устойчивых состояний произвольным образом.

Синхронные D-защелки

Чтобы разрешить ситуацию с неопределенностью SR-защелки (неопределенность возникает в случае, если $S = R = 1$), нужно предотвратить ее возникновение.

На рис. 3.22 изображена схема защелки только с одним входом *D*. Так как входной сигнал в нижний вентиль И всегда является обратным кодом входного сигнала в верхний вентиль И, ситуация, когда оба входа равны 1, никогда не возникает. Когда $D = 1$ и синхронизирующий вход равен 1, защелка переходит в состояние $Q = 1$. Когда $D = 0$ и синхронизирующий вход равен 1, защелка переходит в состояние $Q = 0$. Другими словами, когда синхронизирующий вход равен 1, текущее значение *D* отбирается и сохраняется в защелке. Такая схема, которая называется **синхронной D-защелкой**, представляет собой память объемом один бит. Сохраненное значение всегда доступно на выходе *Q*. Чтобы загрузить в память текущее значение *D*, нужно пустить положительный импульс по линии синхронизирующего сигнала.

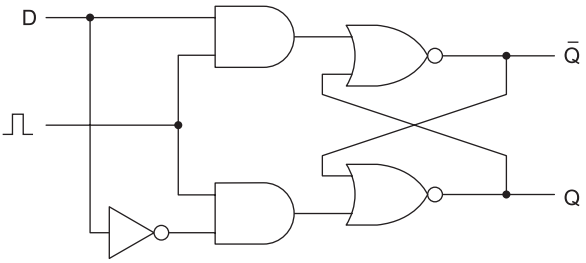


Рис. 3.22. Синхронная D-защелка

Такая схема требует 11 транзисторов. Более сложные схемы (именно они обычно используются на практике) могут хранить один бит всего на 6 транзисторах. Схема остается в устойчивом состоянии до тех пор, пока на нее подается питание (на рисунке не обозначено). Позднее мы рассмотрим схемы, которые быстро забывают состояние, в котором они находятся, — чтобы этого не происходило, им необходимы постоянные «напоминания».

Триггеры

Многие схемы при необходимости выбирают значение на определенной линии в заданный момент времени и запоминают его. В такой схеме, которая называется **триггером** (flip-flop), смена состояния происходит не тогда, когда синхронизирующий сигнал равен 1, а при переходе синхронизирующего сигнала с 0 на 1 (фронт) или с 1 на 0 (спад). Следовательно, длина синхронизирующего импульса не имеет значения, поскольку переходы происходят быстро.

Подчеркнем еще раз отличие между триггером и защелкой. Триггер **запускается перепадом сигнала**, а защелка **запускается уровнем сигнала**. Обратите внимание, что в литературе эти термины часто путают. Многие авторы используют термин «триггер», когда речь идет о защелке, и наоборот[1].

Существуют несколько подходов к проектированию триггеров. Например, если бы существовал способ генерирования очень короткого импульса на фронте синхронизирующего сигнала, этот импульс можно было бы подавать в D-защелку.

[1] В отечественной литературе термин «защелка» (latch) вообще не используется, говорят о триггерах. Однако при этом вводится понятие Т-триггера, который и является «настоящим» триггером. — *Примеч. науч. ред.*

В действительности такой способ существует. Соответствующая схема показана на рис. 3.23, *а*.

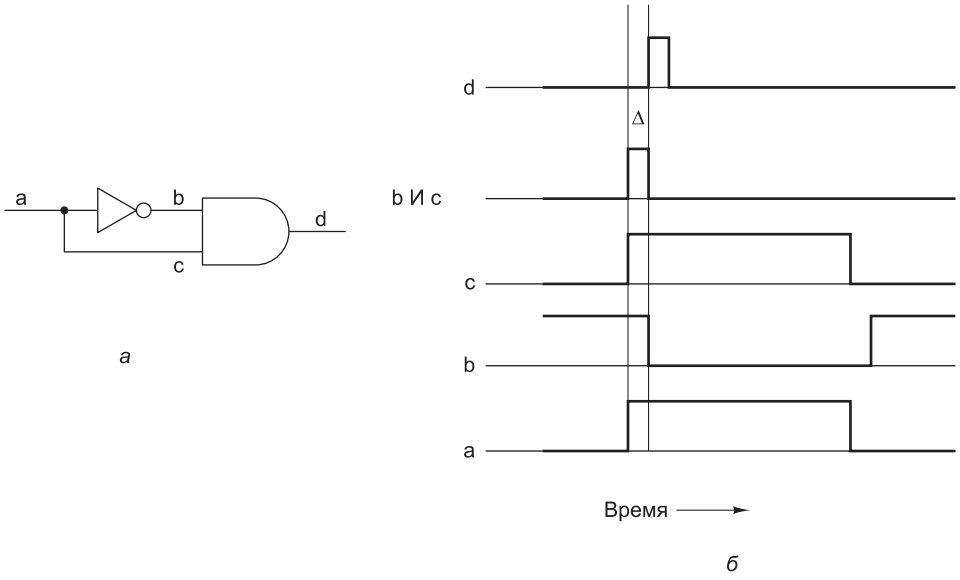


Рис. 3.23. Генератор импульса (*а*); временна́я диаграмма для четырех точек на схеме (*б*)

На первый взгляд может показаться, что выход вентиля И всегда будет нулевым, поскольку функция И от любого сигнала с его инверсией дает 0, но на самом деле ситуация несколько сложнее. При прохождении сигнала через инвертор происходит небольшая, но все-таки не нулевая задержка. Данная схема работает именно благодаря этой задержке. Предположим, мы измеряем напряжение в четырех точках *a*, *b*, *c* и *d*. Входной сигнал в точке *a* представляет собой длинный синхронизирующий импульс (нижний график на рис. 3.23, *б*). Сигнал в точке *b* показан над ним. Отметим, что этот сигнал инвертирован и подается с некоторой задержкой. Время задержки зависит от типа инвертора и обычно составляет несколько наносекунд.

Сигнал в точке *c* тоже подается с задержкой, но эта задержка обусловлена только временем прохождения сигнала (со скоростью света). Если физическое расстояние между точками *a* и *c* составляет, например, 20 микрон, тогда задержка на распространение сигнала равна 0,0001 нс, что, конечно, незначительно по сравнению с временем прохождения сигнала через инвертор. Таким образом, сигнал в точке *c* практически идентичен сигналу в точке *a*.

Когда входные сигналы *b* и *c* подвергаются операции И, в результате получается короткий импульс, длина которого (Δ) равна вентильной задержке инвертора (обычно 5 нс и ниже). Выходной сигнал вентиля И — данный импульс, сдвинутый из-за задержки вентиля И (верхний график на рис. 3.23, *б*). Этот временной сдвиг означает только то, что D-защелка активизируется с определенной задержкой после фронта синхронизирующего импульса. Он никак не влияет на длину импульса. В памяти со временем цикла в 10 нс импульс в 1 нс (который сообщает, когда нужно выбирать линию *D*) достаточно короткий, и в этом случае

полная схема может быть такой, как на рис. 3.24. Следует упомянуть, что такая схема триггера проста для понимания, но на практике обычно используются более сложные триггеры.

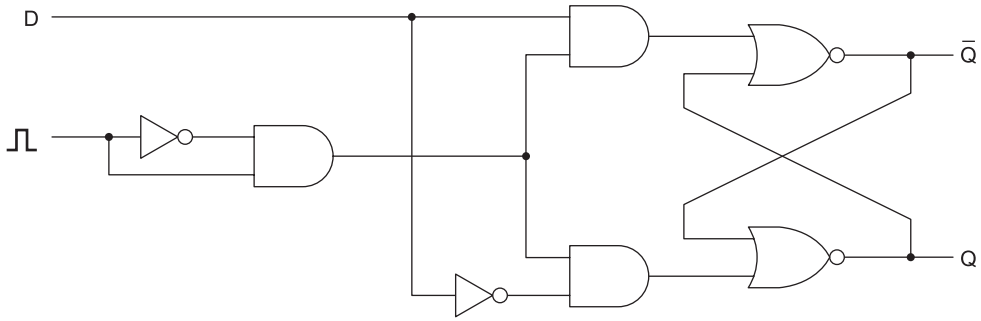


Рис. 3.24. D-триггер

Стандартные обозначения защелок и триггеров показаны на рис. 3.25. На рис. 3.25, *а* изображена защелка, состояние которой загружается тогда, когда синхронизирующий сигнал *CK* (от слова clock) равен 1, в противоположность защелке, изображенной на рис. 3.25, *б*, у которой синхронизирующий сигнал обычно равен 1, но который переходит на 0, чтобы загрузить состояние из линии *D*. На рисунках 3.25, *в*, *г* изображены триггеры. О том, что это триггеры, а не защелки, говорит уголок на синхронизирующем входе. Триггер на рис. 3.25, *в* изменяет состояние на фронте синхронизирующего импульса (переход от 0 к 1), тогда как триггер на рис. 3.25, *г* изменяет состояние на спаде (переход от 1 к 0). Многие (хотя не все) защелки и триггеры также имеют выход $\overline{Q}$, а у некоторых есть два дополнительных входа: Set (установка) или Preset (предварительная установка) и Reset (сброс) или Clear (очистка). Первый вход (Set или Preset) устанавливает $Q = 1$, а второй (Reset или Clear) — $Q = 0$.

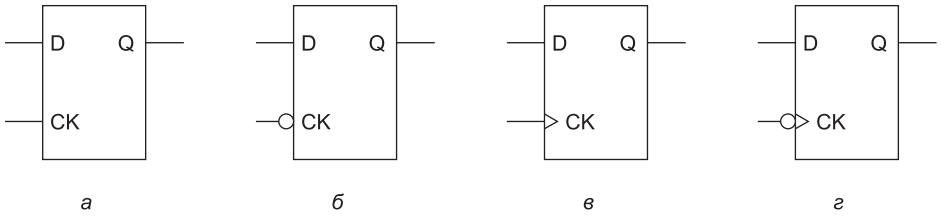


Рис. 3.25. D-защелки и D-триггеры

Регистры

Существуют различные конфигурации триггеров. На рис. 3.26 показано, как восемь триггеров объединяются для формирования 8-разрядного регистра. Регистр получает 8-разрядное входное значение (I_0 –I_7) при изменении синхронизирующего сигнала *CK*. Все синхронизирующие линии связаны с одним входным сигналом *CK*, чтобы при изменении состояния *CK* регистр получал новое 8-разрядное значение данных с входной шины. Сами триггеры того же типа, что и на рис. 3.25, *г*, но инвертирующие входы аннулируются инвертором, связанным

с *CK*, поэтому триггеры запускаются при переходе от 0 к 1. Все восемь сигналов очистки тоже объединены, поэтому когда сигнал сброса *CLR* переходит в состояние 0, все триггеры переходят в состояние 0. Если вам не понятно, почему синхронизирующий сигнал *CK* инвертируется на входе, а затем инвертируется снова в каждом триггере, то ответ прост: входной сигнал не имеет достаточной мощности, чтобы запустить все восемь триггеров; входной инвертор на самом деле используется в качестве усилителя.

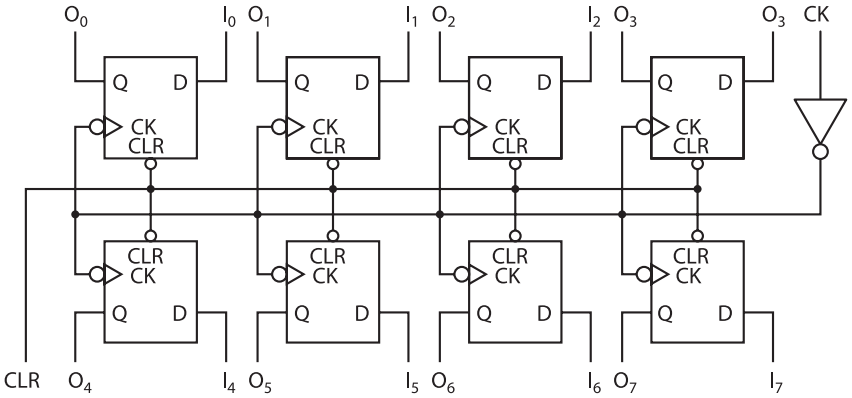


Рис. 3.26. 8-разрядный регистр, построенный из одноразрядных триггеров

8-разрядный регистр может использоваться в качестве структурного элемента для построения регистров большей разрядности. Например, 32-разрядный регистр может быть построен из двух 16-разрядных регистров, для чего следует связать их синхронизирующие линии *CK* и линии сброса *CLR*. Регистры и их применение мы рассмотрим более подробно в главе 4.

Организация памяти

Хотя мы и совершили переход от простой памяти емкостью в 1 бит (см. рис. 3.22) к 8-разрядной памяти (см. рис. 3.26), для построения памяти большого объема потребуется другой способ организации, при котором можно обращаться к отдельным словам. Пример организации памяти, которая удовлетворяет этому критерию, показан на рис. 3.27. Эта память содержит четыре 3-разрядных слова. Каждая операция считывает или записывает целое 3-разрядное слово. Хотя общий объем памяти (12 бит) не на много больше, чем у 8-разрядного триггера, такая память требует меньшего количества выводов, и, что особенно важно, подобная организация применима для построения памяти большого объема. Обратите внимание: количество слов всегда равно степени 2.

Хотя организация памяти, изображенной на рис. 3.27, может на первый взгляд показаться сложной, на самом деле она очень проста благодаря своей регулярной структуре. Микросхема содержит 8 входных линий, в частности 3 входа для данных — I_0, I_1 и I_2; 2 входа для адресов — A_0 и A_1; 3 входа для управления — *CS* (Chip Select — выбор элемента памяти), *RD* (ReaD — чтение, этот сигнал позволяет отличать считывание от записи) и *OE* (Output Enable — разрешение выдачи выходных сигналов), а также 3 выходные линии для данных — O_0, O_1 и O_2.

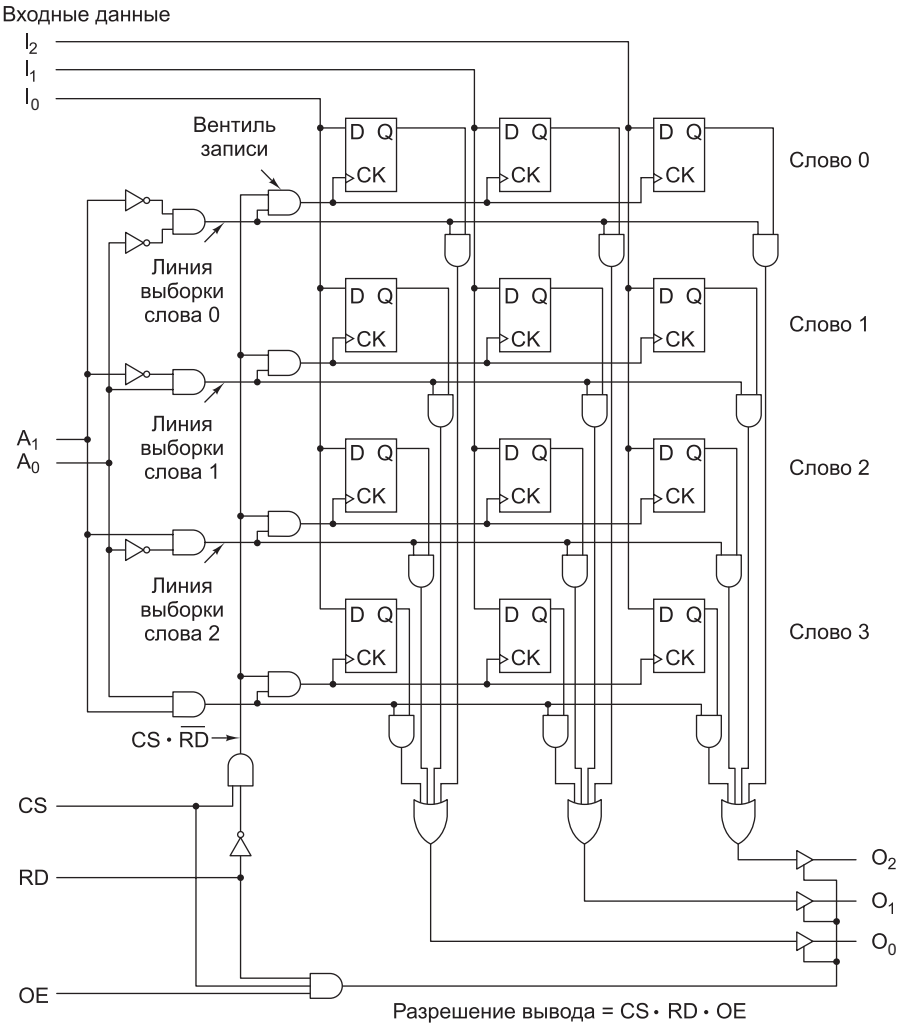


Рис. 3.27. Логическая блок-схема для памяти 4 × 3. Каждый ряд представляет одно из 3-разрядных слов. При считывании и записи всегда считывается или записывается целое слово

Интересно, что такая 12-разрядная память требует меньшего количества выводов, чем 8-разрядный регистр из предыдущего примера. 8-разрядный триггер требует наличия 20 выводов (включая питание и землю), а для 12-разрядной памяти достаточно всего 13, поскольку в отличие от регистра 4 бита памяти совместно используют один выходной сигнал. Состояние адресного входа определяет, каким четырем битам памяти разрешается ввод или вывод значения.

Чтобы выбрать этот блок памяти, внешняя логика должна установить сигнал *CS* в 1, а также установить сигнал *RD* в 1 для чтения и в 0 для записи. Две адресные линии должны указывать, какое из четырех 3-разрядных слов нужно считывать или записывать. При считывании входные линии для данных не ис-

пользуются, а выбранное слово подается на выходные линии для данных. При записи биты, находящиеся на входных линиях для данных, загружаются в выбранное слово памяти; выходные линии при этом не используются.

А теперь давайте посмотрим, как работает память, изображенная на рис. 3.27. Четыре вентиля И для выбора слов в левой части схемы формируют декодер. Входные инверторы расположены так, что каждый вентиль запускается определенным адресом. Каждый вентиль приводит в действие линию выбора слов (для слов 0, 1, 2 и 3). Когда микросхема должна производить запись, вертикальная линия CS·$\overline{\text{RD}}$ получает значение 1, запуская один из четырех вентилей записи. Выбор вентиля зависит от того, какая именно линия выбора слов равна 1. Выходной сигнал вентиля записи приводит в действие все сигналы CK для выбранного слова, загружая входные данные в триггеры для этого слова. Запись производится только в том случае, если сигнал CS равен 1, а RD — 0, при этом записывается только слово, выбранное адресами A_0 и A_1; остальные слова не меняются.

Процесс считывания сходен с процессом записи. Декодирование адреса происходит точно так же, как и при записи. Но в данном случае линия CS·$\overline{\text{RD}}$ принимает значение 0, поэтому все вентили записи блокируются, и ни один из триггеров не меняется. Вместо этого линия выбора слов запускает вентили И, связанные с битами *Q* выбранного слова. Таким образом, выбранное слово передает свои данные в 4-входовые вентили ИЛИ, расположенные в нижней части схемы, а остальные три слова выдают 0. Следовательно, выход вентилей ИЛИ идентичен значению, сохраненному в данном слове. Остальные три слова никак не влияют на выходные данные.

Мы могли бы разработать схему, в которой три вентиля ИЛИ соединялись бы с тремя линиями вывода данных, но это вызвало бы некоторые проблемы. Мы рассматривали линии ввода данных и линии вывода данных как разные линии. На практике же используются одни и те же линии. Если бы мы связали вентили ИЛИ с линиями вывода данных, микросхема пыталась бы выводить данные (то есть задавать каждой линии определенную величину) даже в процессе записи, мешая нормальному вводу данных. По этой причине желательно каким-то образом соединять вентили ИЛИ с линиями вывода данных при считывании и полностью разъединять их при записи. Все, что нам нужно — электронный переключатель, который может устанавливать и разрывать связь за долю наносекунды.

К счастью, такие переключатели существуют. На рис. 3.28, *а* показано символическое изображение так называемого **буферного элемента без инверсии**. Он содержит входную линию для данных, выходную линию для данных и входную линию для управления. Когда управляющий вход равен 1, буферный элемент работает как проводник (рис. 3.28, *б*). Когда управляющий вход равен 0, буферный элемент работает как изолятор (рис. 3.28, *в*), как будто кто-то отрезает выход для данных от остальной части схемы кусачками. Соединение может быть восстановлено за несколько наносекунд, если сделать сигнал управления равным 1.

На рис. 3.28, *г* показан **буферный элемент с инверсией**, который действует как обычный инвертор, когда сигнал управления равен 1, и отделяет выход от остальной части схемы, когда сигнал управления равен 0. Оба буферных элемента представляют собой **устройства с тремя состояниями**, поскольку могут выдавать нулевой сигнал, единичный сигнал или вообще не выдавать никакого сигнала (случай разомкнутой цепи). Буферные элементы, кроме того, усиливают

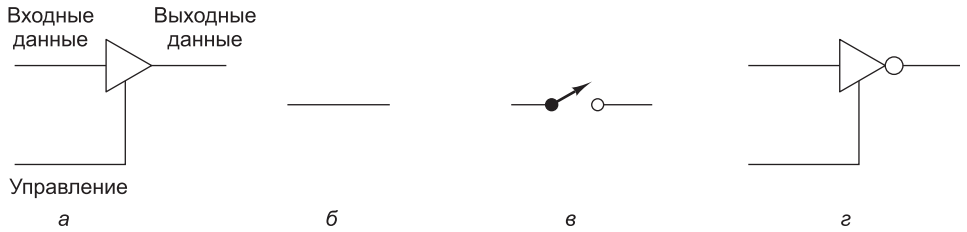


Рис. 3.28. Буферный элемент без инверсии (*а*); представление буферного элемента без инверсии, когда сигнал управления равен 1 (*б*); представление буферного элемента без инверсии, когда сигнал управления равен 0 (*в*); буферный элемент с инверсией (*г*)

сигналы, поэтому они могут справляться с большим количеством сигналов одновременно. Иногда они используются в схемах именно в качестве усилителей, при этом их способность переключения не используется.

Теперь вам уже должно быть понятно, для чего нужны три буферных элемента без инверсии на линиях вывода данных. Когда сигналы *CS*, *RD* и *OE* равны 1, то сигнал разрешения выдачи выходных данных также равен 1, в результате запускаются буферные элементы, и слово помещается на выходные линии. Когда один из сигналов CS, RD и OE равен 0, выходы отсоединяются от остальной части схемы.

Микросхемы памяти

Преимущество памяти, изображенной на рис. 3.27, состоит в том, что подобная структура применима при разработке памяти большого объема. На рисунке показана схема 4 × 3 (для 4-х слов по 3 бита каждое). Чтобы расширить ее до размеров 4 × 8, нужно добавить еще 5 колонок триггеров по 4 триггера в каждой, а также 5 входных и 5 выходных линий. Чтобы перейти от схемы 4 × 3 к схеме 8 × 3, требуется добавить еще четыре ряда триггеров по три триггера в каждом, а также адресную линию A_2. При такой структуре число слов в памяти должно быть степенью двойки для максимальной эффективности, а число битов в слове может быть любым.

Технология изготовления интегральных схем идеально соответствует регулярной структуре микросхем памяти. С развитием технологии число битов, которое можно вместить в одной микросхеме, постоянно растет, обычно в два раза каждые 18 месяцев (закон Мура). С появлением больших микросхем маленькие микросхемы не всегда сразу устаревают, поскольку всегда существует компромисс между емкостью, быстродействием, мощностью, ценой и удобством сопряжения. Обычно самые большие современные микросхемы пользуются огромным спросом и, следовательно, стоят дороже в расчете за один бит, чем микросхемы небольшого размера.

При любом объеме памяти существуют несколько вариантов организации микросхемы. На рис. 3.29 показаны две возможные структуры микросхемы емкостью 4 Мбит: 512 К × 8 и 4096 К × 1 (размеры микросхем памяти обычно даются в битах, а не в байтах, поэтому здесь мы будем придерживаться этого соглашения). На рис. 3.29, *а* можно видеть 19 адресных линий для обращения к одному из 2^{19} байт и 8 линий данных для загрузки или хранения выбранного байта.

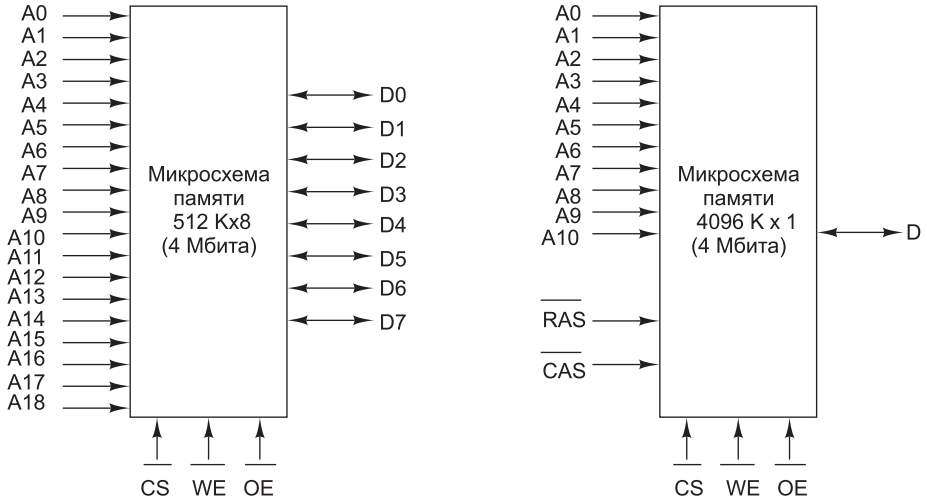


Рис. 3.29. Два способа организации памяти объемом 4 Мбит

Сделаем небольшое замечание по поводу терминологии. На одних выводах высокое напряжение вызывает какое-либо действие, на других остается низкое напряжение. Чтобы избежать путаницы, мы будем употреблять термин **установить сигнал**, когда вызывается какое-то действие, вместо того чтобы говорить, что напряжение повышается или понижается. Таким образом, для одних выводов установка сигнала означает установку единицы, для других — установку нуля. Названия выводов, которые устанавливаются в 0, содержат сверху черту. То есть сигнал CS — это единица, сигнал $\overline{\text{CS}}$ — ноль. Противоположный термин — **сбросить**.

А теперь вернемся к нашей микросхеме. Поскольку обычно компьютер содержит много микросхем памяти, нужен сигнал для выбора необходимой микросхемы, такой, чтобы нужная нам микросхема реагировала на вызов, а остальные нет. Сигнал $\overline{\text{CS}}$ (Chip Select — выбор элемента памяти) используется именно для этой цели. Он устанавливается, чтобы запустить микросхему. Кроме того, нужен способ, чтобы отличать считывания от записи. Сигнал $\overline{\text{WE}}$ (Write Enable — разрешение записи) указывает на то, что данные должны записываться, а не считываться. Наконец, сигнал $\overline{\text{OE}}$ (Output Enable — разрешение вывода) устанавливается для выдачи выходных сигналов. Когда этого сигнала нет, выход отсоединяется от остальной части схемы.

На рис. 3.29, *б* используется другая схема адресации. Микросхема представляет собой матрицу размером 2048 × 2048 однобитных ячеек, что составляет 4 Мбит. Чтобы обратиться к микросхеме, сначала нужно выбрать строку. Для этого 11-разрядный номер этой строки подается на адресные выводы. Затем устанавливается сигнал $\overline{\text{RAS}}$ (Row Address Strobe — строб адреса строки). После этого на адресные выводы подается номер столбца и устанавливается сигнал $\overline{\text{CAS}}$ (Column Address Strobe — строб адреса столбца). Микросхема реагирует на сигнал, принимая или выдавая один бит данных.

Большие микросхемы памяти часто производятся в виде матриц размером $m \times n$, обращение к которым происходит по строкам и столбцам. Такая организация памяти сокращает число необходимых выводов, но, с другой стороны, замедляет обращение к микросхеме, поскольку требуется два цикла адресации: один для строки, другой для столбца. Потеря скорости отчасти компенсируется тем, что в некоторых микросхемах возможна передача адреса строки с последующей передачей нескольких адресов столбцов для обращения к последовательным битам строки.

Много лет назад самые большие микросхемы памяти обычно были устроены так, как показано на рис. 3.29, *б*. Поскольку размер слов увеличился от 8 до 32 бит и выше, использовать подобные микросхемы стало неудобно. Чтобы из микросхем 4096 K × 1 построить память с 32-разрядными словами, требуется 32 микросхемы, работающие параллельно. Эти 32 микросхемы имеют общий объем по крайней мере 16 Мбайт. Если использовать микросхемы 512 K × 8, то потребуется всего 4 микросхемы, но при этом объем памяти составит 2 Мбайт. Чтобы не возиться с 32 микросхемами, большинство производителей выпускают семейства микросхем с длиной слов 4, 8 и 16 бит. Ситуация с 64-разрядными словами, естественно, еще хуже.

Примеры современных микросхем объемом 512 Мбит показаны на рис. 3.30. В каждой такой микросхеме содержится четыре внутренних банка памяти по 128 Мбит; соответственно, для определения банка требуются две линии выбора банка. На микросхеме 32 M × 16, показанной на рис. 3.30, *а*, 13 линий выделено для сигналов $\overline{\text{RAS}}$, 10 для сигналов $\overline{\text{CAS}}$ и 2 линии для выбора банка. Взятые в целом, 25 сигналов обеспечивают возможность адресации 2^{25} внутренних 16-разрядных ячеек. На микросхеме 128 M × 4, изображенной на рис. 3.30, *б*, для сигналов $\overline{\text{RAS}}$ выделено 13 линий, для $\overline{\text{CAS}}$ — 12 линий, для выбора банка —

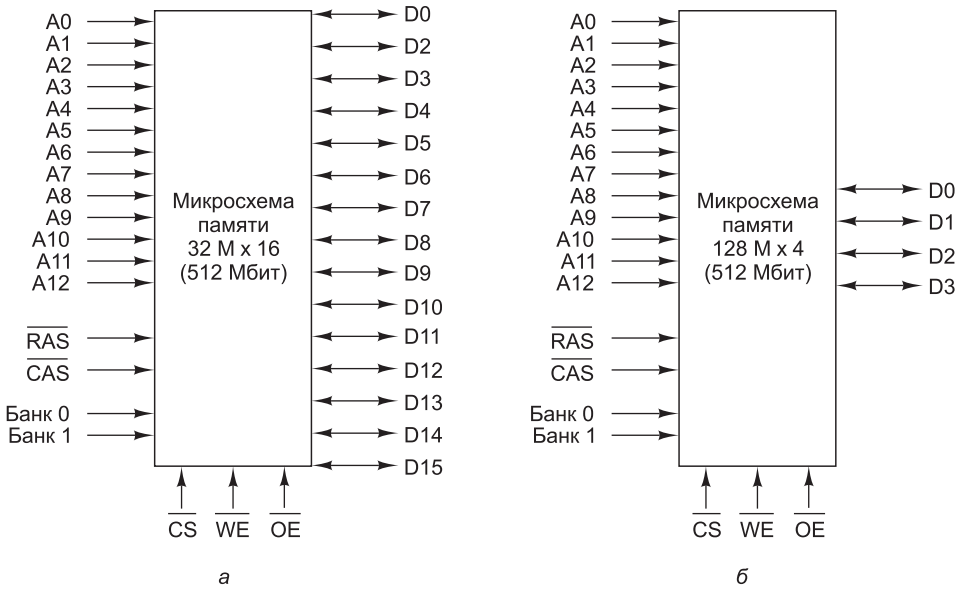


Рис. 3.30. Два способа организации микросхемы памяти объемом 512 Мбит

2 линии. Таким образом, 27 сигналов делают возможной адресацию любой из 2^{27} внутренних 4-разрядных ячеек. Количества строк и столбцов в микросхемах определяются на основании инженерных факторов. Матрица не обязательно должна быть квадратной.

Эти примеры наглядно демонстрируют значимость двух не связанных друг с другом аспектов конструирования микросхем памяти. Первый касается ширины выхода (в битах) — иначе говоря, количества битов (1, 4, 8, 16 и пр.) в выходном сигнале. Второй аспект заключается в способе представления битов адреса; здесь есть два варианта: во-первых, биты адресов могут быть представлены одновременно на разных выводах, во-вторых, может быть последовательное представление строк и столбцов — так, как показано на рис. 3.30. Прежде чем приступать к проектированию микросхемы, специалист должен определиться с обоими аспектами.

ОЗУ и ПЗУ

Все виды памяти, которые мы рассматривали до сих пор, имеют одно общее свойство: они позволяют и записывать, и считывать информацию. Такая память называется **ОЗУ** (оперативное запоминающее устройство), или **RAM** (Random Access Memory — **оперативная память**). Существует два типа ОЗУ: статическое и динамическое. **Статическое ОЗУ** (Static RAM, **SRAM**) конструируется с использованием D-триггеров. Информация в ОЗУ сохраняется на протяжении всего времени, пока к нему подается питание: секунды, минуты, часы и даже дни. Статическое ОЗУ работает очень быстро. Обычно время доступа составляет несколько наносекунд. По этой причине статическое ОЗУ часто используется в качестве кэш-памяти второго уровня.

В **динамическом ОЗУ** (Dynamic RAM, **DRAM**), напротив, триггеры не используются. Динамическое ОЗУ представляет собой массив ячеек, каждая из которых содержит транзистор и крошечный конденсатор. Конденсаторы могут быть заряженными и разряженными, что позволяет хранить нули и единицы. Поскольку электрический заряд имеет тенденцию исчезать, каждый бит в динамическом ОЗУ должен **обновляться** (перезаряжаться) каждые несколько миллисекунд, чтобы предотвратить утечку данных. Поскольку об обновлении должна заботиться внешняя логика, динамическое ОЗУ требует более сложного сопряжения, чем статическое, хотя этот недостаток компенсируется большим объемом.

Поскольку динамическому ОЗУ нужен только один транзистор и один конденсатор на бит (статическому ОЗУ требуется в лучшем случае 6 транзисторов на бит), динамическое ОЗУ имеет очень высокую плотность записи (много битов на одну микросхему). По этой причине основная память почти всегда строится на основе динамических ОЗУ. Однако динамические ОЗУ работают очень медленно (время доступа занимает десятки наносекунд). Таким образом, сочетание кэш-памяти на основе статического ОЗУ и основной памяти на основе динамического ОЗУ соединяет в себе преимущества обоих устройств.

Существует несколько типов динамических ОЗУ. Самый древний тип, который все еще используется, — **FPM** (Fast Page Mode — быстрый постраничный режим). Это ОЗУ представляет собой матрицу битов. Аппаратное обеспечение представляет адрес строки, а затем — адреса столбцов (мы описывали этот

процесс, когда говорили об устройстве памяти, показанном на рис. 3.29, *б*). Благодаря явно передаваемым сигналам память работает асинхронно по отношению к главному тактовому генератору системы.

FPM постепенно замещается памятью **EDO** (Extended Data Output — память с расширенными возможностями вывода)[1], которая позволяет обращаться к памяти еще до того, как закончилось предыдущее обращение. Такой конвейерный режим, хотя и не ускоряет доступ к памяти, повышает пропускную способность, позволяя получить больше слов в секунду.

Память типа FPM и EDO сохраняла актуальность в те времена, когда продолжительность цикла работы микросхем памяти не превышала 12 нс. Впоследствии, с увеличением быстродействия процессоров, сформировалась потребность в более быстрых микросхемах памяти, и тогда на смену асинхронным режимам FPM и EDO пришли **синхронные динамические ОЗУ** (Synchronous DRAM, **SDRAM**). Синхронное динамическое ОЗУ управляется от главного системного тактового генератора. Данное устройство представляет собой гибрид статического и динамического ОЗУ. Основное преимущество синхронного динамического ОЗУ состоит в том, что оно исключает зависимость микросхемы памяти от управляющих сигналов. ЦП сообщает памяти, сколько циклов следует выполнить, а затем запускает ее. Каждый цикл на выходе дает 4, 8 или 16 бит в зависимости от количества выходных строк. Устранение зависимости от управляющих сигналов приводит к ускорению передачи данных между ЦП и памятью.

Следующим этапом в развитии памяти SDRAM стала память **DDR** (Double Data Rate — передача данных с двойной скоростью). Эта технология предусматривает вывод данных как на фронте, так и на спаде импульса, вследствие чего скорость передачи увеличивается вдвое. Например, 8-разрядная микросхема такого типа, работающая с частотой 200 МГц, дает на выходе два 8-разрядных значения 200 миллионов раз в секунду (разумеется, такая скорость удерживается в течение небольшого периода времени); таким образом, теоретически, кратковременная скорость может достигать 3,2 Гбайт/с. Интерфейсы памяти DDR2 и DDR3 обеспечивают дополнительный прирост производительности по сравнению с DDR за счет повышения скорости шины памяти до 533 МГц и 1067 МГц соответственно. На момент издания книги самые быстрые микросхемы DDR3 могли выдавать данные на скорости 17,067 Гбайт/с.

Энергонезависимая память

ОЗУ — не единственный тип микросхем памяти. Во многих случаях данные должны сохраняться даже при отключенном питании (например, если речь идет об игрушках, различных приборах и машинах). Более того, после установки ни программы, ни данные не должны изменяться. Эти требования привели к появлению **ПЗУ** (постоянных запоминающих устройств), или **ROM** (Read-Only Memory — постоянная память). ПЗУ не позволяют изменять и стирать хранящуюся в них информацию (ни умышленно, ни случайно). Данные записываются в ПЗУ в процессе производства. Для этого изготавливается трафарет с определенным набором битов, который накладывается на фоточувствительный

[1] Динамическая память типа EDO практически вытеснила обычную динамическую память, работающую в режиме FPM, в середине девяностых годов. — *Примеч. науч. ред.*

материал, а затем открытые (или закрытые) части поверхности вытравливаются. Единственный способ изменить программу в ПЗУ — поменять всю микросхему.

ПЗУ стоят гораздо дешевле ОЗУ, если заказывать их большими партиями, чтобы оплатить расходы на изготовление трафарета. Однако они не допускают изменений после выпуска с производства, а между подачей заказа на ПЗУ и его выполнением может пройти несколько недель. Чтобы компаниям было проще разрабатывать новые устройства, основанные на ПЗУ, были выпущены **программируемые ПЗУ** (Programmable ROM, **PROM**). В отличие от обычных ПЗУ, их можно программировать в условиях эксплуатации, что позволяет сократить время исполнения заказа. Многие программируемые ПЗУ содержат массив крошечных плавких перемычек. Чтобы пережечь определенную перемычку, нужно выбрать требуемые строку и столбец, а затем приложить высокое напряжение к определенному выводу микросхемы.

Следующая разработка этой линии — **стираемое программируемое ПЗУ** (Erasable PROM, **EPROM**), которое можно не только программировать в условиях эксплуатации, но и стирать с него информацию. Если кварцевое окно в данном ПЗУ подвергать воздействию сильного ультрафиолетового света в течение 15 минут, все биты установятся в 1. Если нужно сделать много изменений во время одного этапа проектирования, стираемые ПЗУ гораздо экономичнее, чем обычные программируемые ПЗУ, поскольку их можно использовать многократно. Стираемые программируемые ПЗУ обычно устроены так же, как статические ОЗУ. Например, микросхема 27C040 имеет структуру, которая показана на рис. 3.30, *а*, а такая структура типична для статического ОЗУ. Интересно, что подобные «древние» микросхемы не вымирают. Становятся дешевле и используются в бюджетных продуктах, для которых критична стоимость. Сейчас одиночные микросхемы 27C040 можно купить дешевле $3, а при большом размере партии они обойдутся значительно дешевле.

Следующий этап — **электронно перепрограммируемое ПЗУ** (Electronically EPROM, **EEPROM**), которое не нужно для этого помещать в специальную камеру, чтобы подвергнуть воздействию ультрафиолетовых лучей — для стирания информации достаточно подать соответствующие импульсы. Кроме того, чтобы перепрограммировать данное устройство, его не нужно вставлять в специальный аппарат для программирования, в отличие от стираемого программируемого ПЗУ. В то же время самые большие электронно перепрограммируемые ПЗУ в 64 раза меньше обычных стираемых ПЗУ, и работают они в два раза медленнее. Электронно перепрограммируемые ПЗУ не могут конкурировать с динамическими и статическими ОЗУ, поскольку работают в 10 раз медленнее, их емкость в 100 раз меньше, и они стоят гораздо дороже. Они используются только в тех ситуациях, когда необходимо сохранять информацию при выключении питания.

Более современный тип электронно перепрограммируемого ПЗУ — **флэш-память**. В отличие от стираемого ПЗУ, которое стирается под воздействием ультрафиолетовых лучей, и от электронно перепрограммируемого ПЗУ, которое стирается по байтам, флэш-память стирается и записывается блоками. Многие изготовители производят небольшие печатные платы, содержащие до 64 Гбайт флэш-памяти. Они используются для хранения изображений в цифровых камерах и для других целей. Как было сказано в главе 2, флэш-память постепенно начинает вытеснять диски, что будет грандиозным шагом вперед, учитывая время

доступа в 50 нс. Флэш-память обеспечивает лучшее время доступа при более низком энергопотреблении; с другой стороны, стоимость одного бита флэш-памяти существенно выше, чем у дисков. Краткое описание различных типов памяти дано в табл. 3.2.

Таблица 3.2. Характеристики различных типов памяти

Тип запоминающего устройства	Категория	Стирание информации	Изменение информации по байтам	Необходимость питания	Применение
SRAM	Чтение и запись	Электрическое	Да	Да	Кэш-память второго уровня
DRAM	Чтение и запись	Электрическое	Да	Да	Основная память (старые модели)
SDRAM	Чтение и запись	Электрическое	Да	Да	Основная память (новые модели)
ROM	Только чтение	Невозможно	Нет	Нет	Устройства большого объема
PROM	Только чтение	Невозможно	Нет	Нет	Устройства небольшого объема
EPROM	Преимущественно чтение	Ультрафиолетовый свет	Нет	Нет	Построение прототипов устройств
EEPROM	Преимущественно чтение	Электрическое	Да	Нет	Построение прототипов устройств
Флэш-память	Чтение и запись	Электрическое	Нет	Нет	Цифровые камеры

FPGA

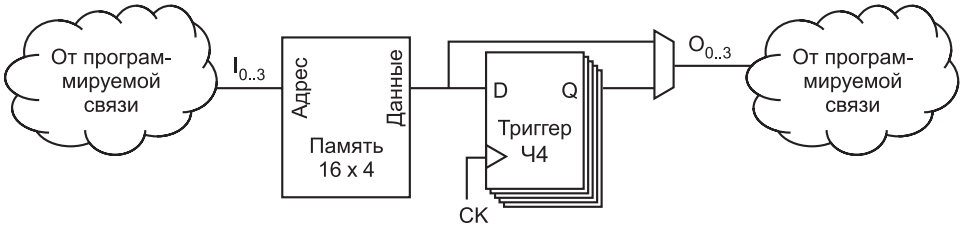
Как упоминалось в главе 1, программируемые вентильные матрицы (**FPGA**) представляют собой микросхемы с программируемой логикой — загрузив в FPGA подходящие конфигурационные данные, можно получить произвольную логическую схему. Главное преимущество FPGA — возможность построения новых аппаратных схем за считанные часы (вместо месяцев, уходящих на производство микросхем). Однако интегрированные схемы не собираются уходить в прошлое — их стоимость в больших партиях существенно ниже, чем у FPGA, они быстрее работают и потребляют меньшую мощность. Однако FPGA из-за своих преимуществ на стадии проектирования часто применяются при построении прототипов устройств и в производстве малыми сериями.

Давайте поближе познакомимся с FPGA и разберемся, как же они используются для реализации широкого диапазона логических схем. Микросхема FPGA состоит из двух основных компонентов, многократно повторяемых в ее архитектуре: таблиц преобразования **LUT** (LookUp Table) и **программируемых связей**.

LUT (рис. 3.31 *а*) представляет собой маленький блок программируемой памяти, который выдает сигнал, передаваемый в регистр (не обязательно), а затем в программируемую связь. Программируемая память используется для создания произвольной логической функции. LUT на рисунке имеет память 16×4, которая способна эмулировать любую логическую схему с четырьмя входными и четырьмя выходными битами. Для программирования LUT в память записываются ответы эмулируемой комбинаторной логики. Иначе говоря, если комбинаторная логика для ввода *X* выдает значение *Y*, то последнее записывается в LUT с индексом *X*.

Пример на рис. 3.31, *б* показывает, как один 4-разрядный блок LUT реализует 3-разрядный счетчик со сбросом. Счетчик из приведенного значения прибавляет к своему текущему значению 1 (сложение по модулю 4), пока не будет получен сигнал сброса *CLR*; в этом случае счетчик сбрасывается в 0.

В реализации счетчика из этого примера четыре верхних элемента LUT заполняются нулями. Они обеспечивают вывод 0 при сбросе счетчика. Таким образом, старший бит LUT (I_3) представляет входной сигнал сброса (*CLR*). У остальных элементов LUT значение с индексом $I_{0..3}$ содержит результат операции (I + 1) по модулю 4. Для завершения этой схемы выходной сигнал $O_{0..3}$ должен быть соединен через программируемую связь с внутренним входным сигналом $I_{0..3}$.



а

Соответствие сигналов

FPGA	Counter
I_3	CLR
$O_{2..0}$	$O_{2..0}$
CK	CK

Addr	Data
0	1
1	2
2	3
3	0

Addr	Data
4	0
5	0
6	0
7	0

б

Рис. 3.31. Таблица преобразования FPGA (*а*); конфигурация LUT для создания 3-разрядного счетчика со сбросом (*б*)

Чтобы лучше понять устройство счетчика FPGA со сбросом, рассмотрим его работу. Если, например, текущее состояние счетчика равно 2, а сигнал сброса (*CLR*) не установлен, то входной адрес LUT будет равен 2, что приведет к выводу в триггер 3. Если в этом состоянии будет установлен сигнал сброса (*CLR*), то на вход LUT поступит 6, в результате чего следующее состояние будет равно 0.

В общем и целом кажется, что перед нами всего лишь хитроумный способ создания счетчика со сбросом. В самом деле, схема с цепью инкремента и сигналом сброса на триггерах будет более компактной, быстрой и расходующей меньше энергии. Главное преимущество схемы на базе FPGA заключается в том, что ее можно построить за час дома, тогда как более эффективную специализированную схему придется производить в кремнии, на что уйдет месяц и более.

Чтобы использовать FPGA, необходимо создать описание схемы или программу на языке описания оборудования (язык программирования, используемый для описания аппаратных структур). Описание обрабатывается синтезатором, который связывает схему с конкретной архитектурой FPGA. Одна из проблем с использованием FPGA заключается в том, что потребности конкретной схемы часто не укладываются в возможности FPGA. FPGA производятся с переменным числом LUT, при этом увеличение количества последних приводит к возрастанию стоимости. Как правило, если результат не укладывается в рамки требований, приходится либо упрощать схему, либо отказываться от части функциональности, либо приобретать большую (и более дорогую) матрицу FPGA. Очень большие схемы могут превосходить возможности даже самых крупных FPGA, в этом случае проектировщику приходится объединять в схеме несколько FPGA; задача безусловно усложняется, но по-прежнему остается намного проще проектирования полноценной специализированной интегральной схемы.

Микросхемы процессоров и шины

Вооружившись информацией о микросхемах, тактовых генераторах и микросхемах памяти, мы можем сложить все составные части вместе и начать изучение целых систем. В этом разделе сначала мы рассмотрим процессоры на цифровом логическом уровне, включая **цоколевку** (то есть назначение сигналов на различных выводах). Поскольку центральные процессоры тесно связаны с шинами, которые они используют, мы также кратко изложим основные принципы разработки шин. В следующих разделах приводятся подробные примеры центральных процессоров, их шин и взаимодействий между ними.

Микросхемы процессоров

Все современные процессоры помещаются на одной микросхеме, благодаря чему их взаимодействия с остальными частями системы становятся четко определенными. Каждая микросхема процессора содержит набор выводов, через которые происходит обмен информацией с внешним миром. Одни выводы передают сигналы от центрального процессора, другие принимают сигналы от других компонентов, третьи делают то и другое. Изучив функции всех выводов, мы сможем узнать, как процессор взаимодействует с памятью и устройствами ввода-вывода на цифровом логическом уровне.

Выводы микросхемы центрального процессора можно подразделить на три типа: адресные, информационные и управляющие. Эти выводы связаны с соответствующими выводами на микросхемах памяти и микросхемах устройств ввода-вывода через набор параллельных проводов (так называемую шину). Чтобы вызвать команду, центральный процессор сначала посылает в память адрес этой команды по адресным выводам. Затем он задействует одну или несколько линий управления, чтобы сообщить памяти, что ему нужно (например, прочитать слово). Память выдает ответ, помещая требуемое слово на информационные выводы процессора и посылая сигнал о том, что это сделано. Когда центральный процессор получает этот сигнал, он считывает слово и выполняет вызванную команду.

Команда может требовать чтения или записи слов, содержащих данные. В этом случае весь процесс повторяется для каждого дополнительного слова. Как происходит процесс чтения и записи, мы подробно рассмотрим далее. А пока важно понять, что центральный процессор обменивается информацией с памятью и устройствами ввода-вывода, подавая сигналы на выводы и принимая сигналы на входы. Другого способа обмена информацией не существует.

Число адресных выводов и число информационных выводов — два ключевых параметра, которые определяют производительность процессора. Микросхема, содержащая m адресных выводов, может обращаться к 2^m ячейкам памяти. Обычно m равно 16, 32 или 64. Микросхема, содержащая n информационных выводов, может считывать или записывать n-разрядное слово за одну операцию. Обычно n равно 8, 32 или 64. Центральному процессору с 8 информационными выводами понадобится 4 операции, чтобы считать 32-разрядное слово, тогда как процессор, имеющий 32 информационных вывода, может сделать ту же работу в рамках одной операции. Следовательно, микросхема с 32 информационными выводами работает гораздо быстрее, но и стоит гораздо дороже.

Помимо адресных и информационных выводов, каждый процессор содержит управляющие выводы. Эти выводы позволяют регулировать и синхронизировать поток данных к процессору и от него, а также выполнять другие функции. Все процессоры содержат выводы для питания (обычно +1,2 В или +1,5 В), заземления и синхронизирующего сигнала (меандра). Остальные выводы разнятся от процессора к процессору. Тем не менее управляющие выводы можно разделить на несколько основных категорий:

- управление шиной;
- прерывания;
- арбитраж шины;
- сигналы сопроцессора;
- состояние;
- разное.

Далее мы кратко опишем каждую из этих категорий, а когда мы будем рассматривать микросхемы Intel Core i7, TI OMAP4430 и Atmel ATmega168, дадим более подробную информацию. Схема типичного центрального процессора, в котором используются эти типы сигналов, изображена на рис. 3.32.

Выводы управления шиной по большей части представляют собой выходы из центрального процессора в шину (и, следовательно, входы в микросхем памяти и микросхем устройств ввода-вывода). Они позволяют сообщить, что процессор хочет считать информацию из памяти или записать информацию в память или сделать что-нибудь еще.

Выводы прерывания — это входы из устройств ввода-вывода в процессор. В большинстве систем процессор может дать сигнал устройству ввода-вывода начать операцию, а затем приступить к какому-нибудь другому действию, пока устройство ввода-вывода выполняет свою работу. Когда устройство ввода-вывода ее завершит, контроллер ввода-вывода посылает сигнал на один из выводов прерывания, чтобы прервать работу процессора и заставить его обслужить устройство ввода-вывода (например, проверить ошибки ввода-вывода). Некоторые процессоры содержат вывод для подтверждения сигнала прерывания.

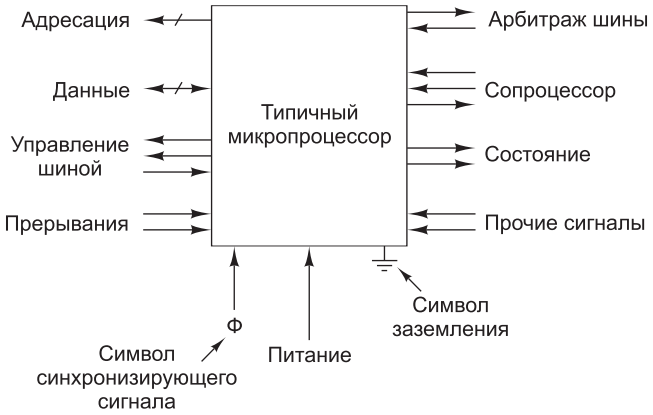


Рис. 3.32. Цоколевка типичного центрального процессора. Стрелками обозначены входные и выходные сигналы, а короткими диагональными линиями — наличие нескольких выводов данного типа. Число этих выводов зависит от модели процессора

Выводы арбитража шины нужны для регулировки потока информации в шине, то есть для исключения таких ситуаций, когда два устройства пытаются воспользоваться шиной одновременно. В плане арбитража центральный процессор считается просто одним из устройств.

Некоторые центральные процессоры могут работать с различными сопроцессорами (например, с графическими процессорами, процессорами для обработки вещественных данных и т. п.). Чтобы обеспечить обмен информацией между процессором и сопроцессором, используются специальные выводы.

Помимо этих выводов, у некоторых процессоров есть дополнительные выводы. Одни из них выдают или принимают информацию о состоянии, другие нужны для перезагрузки компьютера, третьи призваны обеспечивать совместимость со старыми микросхемами устройств ввода-вывода.

Компьютерные шины

Шина — это несколько проводников, соединяющих несколько устройств. Шины можно разделить на категории в соответствии с выполняемыми функциями. Они могут быть внутренними по отношению к процессору и служить для передачи данных в АЛУ и из АЛУ, а могут быть внешними по отношению к процессору и связывать процессор с памятью или устройствами ввода-вывода. Каждый тип шины обладает определенными свойствами и к каждому из них предъявляются определенные требования. В этом и следующих подразделах мы сосредоточимся на шинах, которые связывают центральный процессор с памятью и устройствами ввода-вывода. В следующей главе мы подробно рассмотрим внутренние шины процессора.

Первые персональные компьютеры имели одну внешнюю шину, которая называлась **системной**. Она состояла из нескольких медных проводов (от 50 до 100), которые встраивались в материнскую плату. На материнской плате на одинаковых расстояниях друг от друга находились разъемы для микросхем памяти и устройств ввода-вывода. Современные персональные компьютеры обычно содержат специальную шину между центральным процессором и памятью и по

крайней мере еще одну шину для устройств ввода-вывода. На рис. 3.33 изображена система с одной шиной памяти и одной шиной ввода-вывода.

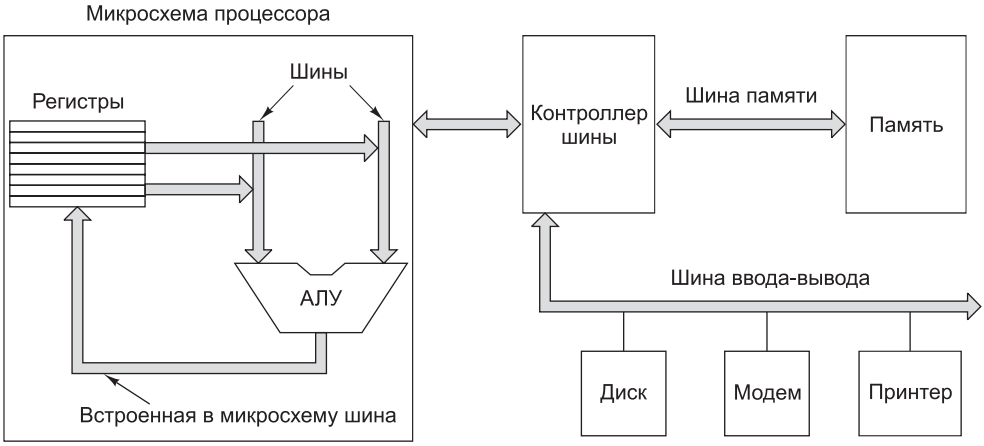


Рис. 3.33. Компьютерная система с несколькими шинами

В литературе шины обычно изображаются в виде жирных стрелок, как показано на этом рисунке. Разница между жирной стрелкой и нежирной стрелкой, через которую проходит короткая диагональная линия с указанием числа битов, небольшая. Когда тип всех битов одинаков, например все адресные или все информационные, рисуется обычная стрелка. Когда включаются адресные линии, линии данных и управления, используется жирная стрелка.

Хотя разработчики процессоров могут использовать любой тип шины для микросхемы, должны быть введены четкие правила о том, как работает шина; и все устройства, связанные с шиной, должны подчиняться этим правилам, чтобы платы, которые выпускаются сторонними производителями, подходили к системной шине. Эти правила называются **протоколом шины**. Кроме того, должны существовать определенные технические требования, чтобы платы от сторонних производителей подходили к направляющим для печатных плат и имели разъемы, соответствующие материнской плате механически, с точки зрения напряжений, синхронизации и т. д. Некоторые шины не имеют механических спецификаций, потому что они спроектированы для использования только с интегральными схемами — например, для соединения компонентов в однокристальных системах (SoC).

Существует целый ряд шин, широко используемых в компьютерном мире, например: Omnibus (PDP-8), Unibus (PDP-11), Multibus (8086), VME (оборудование для физической лаборатории), IBM PC (PC/XT), ISA (PC/AT), EISA (80386), Microchannel (PS/2), Nubus (Macintosh), PCI (различные персональные компьютеры), SCSI (различные персональные компьютеры и рабочие станции), Universal Serial Bus (современные персональные компьютеры), FireWire (бытовая электроника). Может быть, все стало бы намного проще, если бы все шины, кроме одной или двух, исчезли с поверхности земли. К сожалению, стандартизация в этой области кажется очень маловероятной, поскольку во все эти несовместимые системы уже вложено слишком много средств.

Начнем с того, как работают шины. Некоторые устройства, соединенные с шиной, являются активными и могут инициировать передачу информации по шине, тогда как другие являются пассивными и ждут запросов. Активное устройство называется **задающим**, пассивное — **подчиненным**. Когда центральный процессор требует от контроллера диска считать или записать блок информации, центральный процессор действует как задающее устройство, а контроллер диска — как подчиненное. Контроллер диска может действовать как задающее устройство, когда он командует памяти принять слова, которые считал с диска. Несколько типичных комбинаций задающего и подчиненного устройств перечислены в табл. 3.3. Память ни при каких обстоятельствах не может быть задающим устройством.

Таблица 3.3. Примеры задающих и подчиненных устройств

Задающее устройство	**Подчиненное устройство**	**Пример**
Центральный процессор	Память	Вызов команд и данных
Центральный процессор	Устройство ввода-вывода	Инициализация передачи данных
Центральный процессор	Сопроцессор	Передача команды от процессора к сопроцессору
Устройство ввода-вывода	Память	Прямой доступ к памяти
Сопроцессор	Центральный процессор	Вызов сопроцессором операндов из центрального процессора

Двоичным сигналам, которые выдают устройства компьютера, часто не хватает мощности для активизации шины, особенно если она достаточно длинная и если к ней подсоединено много устройств. По этой причине большинство задающих устройств шины обычно связаны с ней через микросхему, которая называется **драйвером шины** и по существу является цифровым усилителем. Сходным образом большинство подчиненных устройств связаны с шиной **приемником шины**. Для устройств, которые могут быть и задающим, и подчиненным устройством, используется **приемопередатчик**, или **трансивер**, **шины**. Эти микросхемы, предназначенные для взаимодействия с шиной, часто являются устройствами с тремя состояниями, что дает им возможность отсоединяться, когда они не нужны. Иногда они подключаются через **открытый коллектор**, что дает сходный эффект. Когда одно или несколько устройств на открытом коллекторе требуют доступа к шине в одно и то же время, результатом является булева операция ИЛИ над всеми этими сигналами. Такое соглашение называется **монтажным ИЛИ**. В большинстве шин одни линии являются устройствами с тремя состояниями, а другие, которым требуется свойство монтажного ИЛИ, — открытым коллектором.

Как и процессор, шина имеет адресные, информационные линии и управляющие линии. Тем не менее между выводами процессора и сигналами шины может и не быть взаимно однозначного соответствия. Например, некоторые процессоры содержат три вывода, которые выдают сигнал чтения из памяти или записи в память, чтения устройства с ввода-вывода, записи на устройство ввода-вывода или выполнения какой-либо другой операции. Обычная шина может содержать одну линию для чтения из памяти, вторую — для записи в память, третью — для

чтения с устройства ввода-вывода, четвертую — для записи на устройство ввода-вывода и т. д. Тогда связывать процессор с такой шиной должна микросхема-декодер, призванная преобразовывать 3-разрядный кодированный сигнал в отдельные сигналы, которые могут управлять линиями шины.

Проектирование и принципы действия шин — это достаточно сложные вопросы, и по этому поводу написан ряд книг [Anderson et al., 2004; Solari and Willse, 2004]. Принципиальными вопросами в разработке являются ширина шины, синхронизация шины, арбитраж шины и функционирование шины. Все эти параметры существенно влияют на пропускную способность шины. В следующих четырех подразделах мы рассмотрим каждый из них.

Ширина шины

Ширина (количество адресных линий) шины — самый очевидный параметр при проектировании. Чем больше адресных линий содержит шина, тем к большему объему памяти может обращаться процессор. Если шина содержит n адресных линий, тогда процессор может использовать ее для обращения к 2^n различным ячейкам памяти. Для памяти большой емкости необходимо много адресных линий. Вроде бы все просто.

Проблема заключается в том, что для широких шин требуется больше проводов, чем для узких. Они занимают больше физического пространства (например, на материнской плате) и для них нужны разъемы большего размера. Все эти факторы делают шину дорогостоящей. Следовательно, необходим компромисс между максимальным объемом доступной памяти и стоимостью системы. Система с шиной, содержащей 64 адресные линии, и памятью в 2^{32} байт будет стоить дороже, чем система с шиной, содержащей 32 адресные линии, и такой же памятью в 2^{32} байт. Дальнейшее расширение не бесплатное.

Многие разработчики систем оказались недальновидными, что привело к неприятным последствиям. Первая модель IBM PC содержала процессор 8088 и 20-разрядную адресную шину (рис. 3.34, *а*). Шина позволяла обращаться к 1 Мбайт памяти.

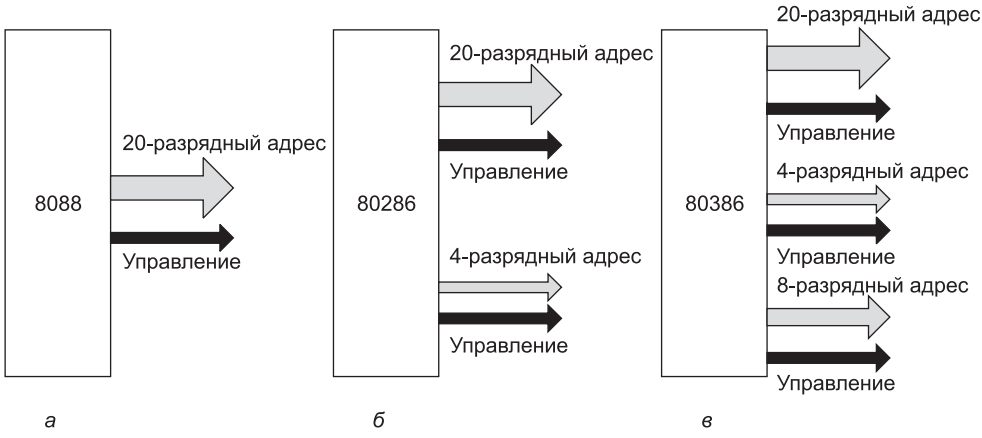


Рис. 3.34. Расширение адресной шины с течением времени

Когда появился следующий процессор (80286), компания Intel решила увеличить адресное пространство до 16 Мбайт, поэтому пришлось добавить еще 4 линии (не нарушая изначальные 20 по причинам совместимости с предыдущими версиями), как показано на рис. 3.34, *б*. К сожалению, пришлось также добавить управляющие линии для новых адресных линий. Когда появился процессор 80386, было добавлено еще 8 адресных линий и, естественно, несколько управляющих линий, как показано на рис. 3.34, *в*. В результате получилась шина EISA. Однако архитектура получилась куда более запутанной, чем если бы с самого начала использовались 32 линии.

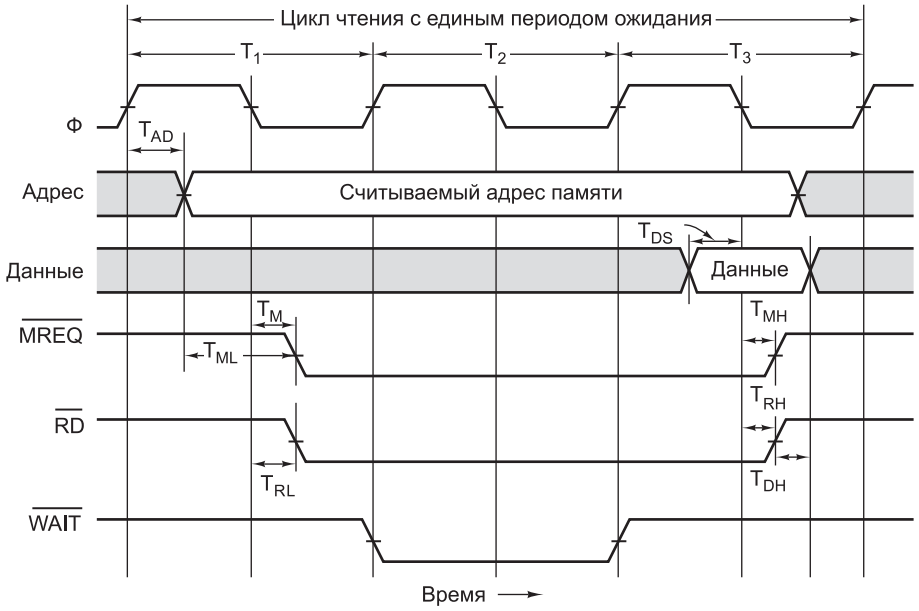
С течением времени увеличивается не только число адресных линий, но и число информационных линий, хотя это происходит по другой причине. Можно увеличить пропускную способность шины двумя способами: сократить время цикла шины (сделать большее количество передач в секунду) или увеличить ширину шины данных (то есть увеличить количество битов, передаваемых за цикл). Можно повысить скорость работы шины, но сделать это довольно сложно, поскольку сигналы на разных линиях передаются с разной скоростью (это явление называется **расфазировкой шины**). Чем быстрее работает шина, тем больше расфазировка.

При увеличении скорости работы шины возникает еще одна проблема: в этом случае она становится несовместимой с предыдущими версиями. Старые платы, разработанные для более медленной шины, не могут работать с новой. Такая ситуация невыгодна для владельцев и производителей старых плат. Поэтому обычно для увеличения производительности просто добавляются новые линии, как показано на рис. 3.34. Как вы понимаете, в этом тоже есть свои недостатки. Компьютер IBM PC и его преемники, например, начали с 8 информационных линий, затем перешли к 16, потом — к 32 линиям, и все это в одной и той же шине.

Чтобы обойти эту проблему, разработчики иногда отдают предпочтение **мультиплексной шине**. В этой шине нет разделения на адресные и информационные линии. В ней может быть, например, 32 линии и для адресов, и для данных. Сначала эти линии используются для адресов, затем — для данных. Чтобы записать информацию в память, нужно сначала передавать в память адрес, а потом — данные. В случае с отдельными линиями адреса и данные могут передаваться вместе. Объединение линий сокращает ширину и стоимость шины, но система работает при этом медленнее. Поэтому проектировщикам приходится взвешивать все за и против, прежде чем сделать выбор.

Синхронизация шины

Шины можно разделить на две категории в зависимости от их синхронизации. **Синхронная шина** содержит линию, которая запускается кварцевым генератором. Сигнал на этой линии представляет собой меандр с частотой обычно от 5 до 133 МГц. Любое действие шины занимает целое число так называемых **циклов шины**. **Асинхронная шина** не содержит задающего генератора. Циклы шины могут быть произвольными и не обязательно одинаковыми для всех пар устройств. Далее мы рассмотрим каждый тип шины отдельно.



а

Символ	Значение	Минимум	Максимум	Единицы измерения
T_{AD}	Задержка выдачи адреса		4	нс
T_{ML}	Промежуток между стабилизацией адреса и установкой сигнала $\overline{MREQ}$	2		нс
T_{M}	Промежуток между спадом синхронизирующего сигнала в цикле T_1 и установкой сигнала $\overline{MREQ}$		3	нс
T_{RL}	Промежуток между спадом синхронизирующего сигнала в цикле T_1 и установкой сигнала $\overline{RD}$		3	нс
T_{DS}	Период передачи данных до спада синхронизирующего сигнала	2		нс
T_{MH}	Промежуток между спадом синхронизирующего сигнала в цикле T_3 и сбросом сигнала $\overline{MREQ}$		3	нс
T_{RH}	Промежуток между спадом синхронизирующего сигнала в цикле T_3 и сбросом сигнала $\overline{RD}$		3	нс
T_{DH}	Период продолжения передачи данных с момента сброса сигнала $\overline{RD}$	0		нс

б

Рис. 3.35. Временна́я диаграмма процесса считывания на синхронной шине (*а*); некоторые временны́е характеристики процесса считывания на синхронной шине (*б*)

Синхронные шины

В качестве примера того, как работает синхронная шина, рассмотрим временну́ю диаграмму на рис. 3.35. В этом примере мы будем использовать задающий генератор на 100 МГц, который дает цикл шины в 10 нс. Хотя может показаться, что шина работает медленно по сравнению с процессорами на 3 ГГц и выше, не многие современные шины работают быстрее. Например, популярная шина PCI работает с частотой 33 МГц или 66 МГц, а улучшенная (но ныне не используемая) шина PCI-X работала с частотой 133 МГц. О причинах такой низкой скорости современных шин уже рассказывалось: к ним можно отнести такие технические проблемы, как расфазировка шины и необходимость обеспечения совместимости.

В своем примере мы предполагаем, что считывание информации из памяти занимает 15 нс с момента установки адреса. Как мы скоро увидим, для чтения слова понадобится три цикла шины. Первый цикл начинается на фронте отрезка T_1, а третий заканчивается на фронте отрезка T_4, как показано на рис. 3.35. Отметим, что ни один из фронтов и спадов не нарисован вертикальным, потому что ни один электрический сигнал не может изменять свое значение за нулевое время. В нашем примере мы предполагаем, что для изменения сигнала требуется 1 нс. Генератор и линии адреса и данных, а также линии $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, $\overline{\text{WAIT}}$ показаны в том же масштабе времени.

Начало T_1 определяется фронтом генератора. За время T_1 центральный процессор помещает адрес нужного слова на адресные линии. Поскольку адрес представляет собой не одно значение (в отличие от генератора), мы не можем показать его в виде одной линии на схеме. Вместо этого мы показали его в виде двух линий с пересечениями там, где этот адрес меняется. Серый цвет на схеме показывает, что в этот момент не важно, какое значение принял сигнал. Используя то же соглашение, мы видим, что содержание линий данных не имеет значения до отрезка T_3.

После того как у адресных линий появляется возможность приобрести новое значение, устанавливаются сигналы $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$. Первый указывает, что осуществляется доступ к памяти, а не к устройству ввода-вывода, а второй — что осуществляется чтение, а не запись. Поскольку после установки адреса считывание информации из памяти занимает 15 нс (часть первого цикла), память не может передать требуемые данные за период T_2. Чтобы центральный процессор не ожидал поступления данных, память устанавливает сигнал $\overline{\text{WAIT}}$ в начале отрезка T_2. Это означает ввод **периодов ожидания** (дополнительных циклов шины) до тех пор, пока память не сбросит сигнал $\overline{\text{WAIT}}$. В нашем примере вводится один период ожидания (T_2), поскольку память работает слишком медленно. В начале отрезка T_3, когда есть уверенность в том, что память получит данные в течение текущего цикла, сигнал $\overline{\text{WAIT}}$ сбрасывается.

Во время первой половины отрезка T_3 память помещает данные на информационные линии. На спаде отрезка T_3 центральный процессор стробирует (то есть считывает) информационные линии, сохраняя их значения во внутреннем регистре. Считав данные, центральный процессор сбрасывает сигналы $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$. В случае необходимости на следующем фронте может начаться еще один цикл памяти. Эта последовательность может повторяться бесконечно.

В хронометражной спецификации на рис. 3.35, *б* используются 8 условных обозначений. T_{AD}, например, — это временной интервал между фронтом T_1 и установкой адресных линий. В соответствии с требованиями синхронизации $T_{AD} \leq 4$ нс. Это значит, что производитель процессора гарантирует, что во время любого цикла считывания центральный процессор сможет выдать требуемый адрес в пределах 4 нс от середины фронта T_1.

Условия синхронизации также требуют, чтобы данные поступали на информационные линии по крайней мере за 2 нс (T_{DS}) до спада T_3, чтобы дать данным время установиться до того, как процессор начнет их стробировать. Сочетание ограничений на T_{AD} и T_{DS} означает, что в худшем случае в распоряжении памяти будет только 25 — 4 — 2 = 19 нс с момента появления адреса и до момента, когда нужно выдавать данные. Поскольку достаточно 10 нс, память даже в самом худшем случае может всегда ответить за период T_3. Если памяти для считывания требуется 20 нс, то необходимо ввести второй период ожидания, и тогда память ответит в течение T_4.

Требования синхронизации гарантируют, что адрес будет установлен по крайней мере за 2 нс до того, как появится сигнал $\overline{\text{MREQ}}$. Это время может быть важно в том случае, если $\overline{\text{MREQ}}$ инициирует выбор элемента памяти, поскольку некоторые типы памяти требуют определенного времени на установку адреса до выбора элемента памяти. Ясно, что разработчику системы не следует выбирать микросхему памяти, которой нужно 3 нс на подготовку.

Ограничения на T_M и T_{RL} означают, что сигналы $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$ будут установлены в пределах 3 нс от спада T_1. В худшем случае у микросхемы памяти после установки сигналов $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$ останется всего 10 + 10 — 3 — 2 = 15 нс на передачу данных по шине. Это ограничение вводится дополнительно по отношению к интервалу в 15 нс и не зависит от него.

Интервалы T_{MH} и T_{RH} определяют, сколько времени требуется на отмену сигналов $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$ после того, как данные стробированы. Наконец, интервал T_{DH} определяет, сколько времени память должна держать данные на шине после снятия сигнала $\overline{\text{RD}}$. В нашем примере при данном процессоре память может удалить данные с шины, как только сбрасывается сигнал $\overline{\text{RD}}$; в случае других процессоров данные могут сохраняться еще некоторое время.

Необходимо подчеркнуть, что наш пример представляет собой весьма упрощенную версию реальных временны́х ограничений. В действительности таких ограничений гораздо больше. Тем не менее этот пример наглядно демонстрирует, как работает синхронная шина.

Отметим, что сигналы управления могут задаваться низким или высоким напряжением. Что является более удобным в каждом конкретном случае, должен решать разработчик, хотя, по существу, выбор произволен. Такую свободу выбора можно назвать «аппаратным» аналогом ситуации, при которой программист может представить свободные дисковые блоки в битовом отображении как в виде нулей, так и в виде единиц.

Асинхронные шины

Хотя использовать синхронные шины благодаря дискретным временны́м интервалам достаточно удобно, здесь все же есть некоторые проблемы. Например, если процессор и память способны закончить передачу за 3,1 цикла, они вынуждены продлить ее до 4,0 цикла, поскольку неполные циклы запрещены.

Еще хуже то, что если однажды был выбран определенный цикл шины и в соответствии с ним разработана память и карты ввода-вывода, то в будущем трудно делать технологические усовершенствования. Например, предположим, что через несколько лет после выпуска системы, изображенной на рис. 3.35, появилась новая память с временем доступа не в 15, а в 8 нс. Это время позволяет избавиться от периода ожидания и увеличить скорость работы машины. А теперь представим, что появилась память с временем доступа в 4 нс. При этом улучшения производительности уже не будет, поскольку в данной разработке минимальное время чтения — 2 цикла.

Если синхронная шина соединяет ряд устройств, одни из которых работают быстро, а другие медленно, шина подстраивается под самое медленное устройство, а более быстрые не могут использовать свой потенциал полностью.

По этой причине были разработаны асинхронные шины, то есть шины без задающего генератора (рис. 3.36). Работа асинхронной шины не привязывается к генератору. Когда задающее устройство устанавливает адрес, сигнал $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ или любой другой требуемый сигнал, он выдает специальный синхронизирующий сигнал $\overline{\text{MSYN}}$ (Master SYNchronization). Когда подчиненное устройство получает этот сигнал, оно начинает выполнять свою работу настолько быстро, насколько это возможно. Когда работа заканчивается, подчиненное устройство выдает сигнал $\overline{\text{SSYN}}$ (Slave SYNchronization).

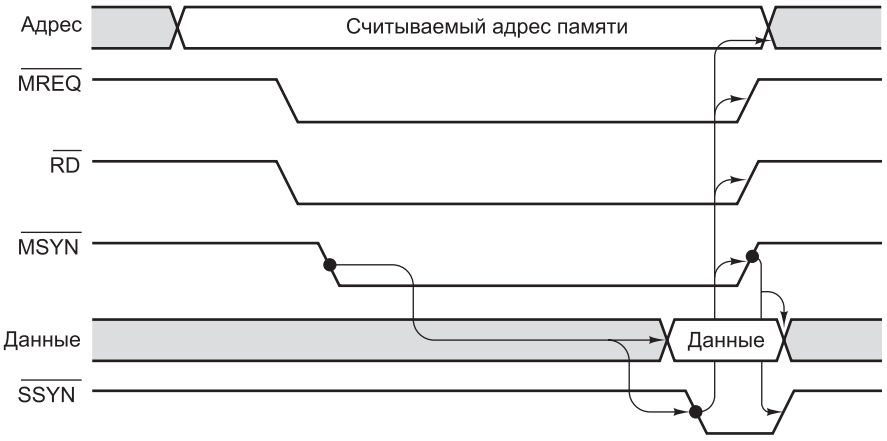


Рис. 3.36. Работа асинхронной шины

Сигнал $\overline{\text{SSYN}}$ сообщает задающему устройству, что данные доступны. Он фиксирует их, а затем сбрасывает адресные линии вместе с сигналами $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ и $\overline{\text{MSYN}}$. Сброс сигнала $\overline{\text{MSYN}}$ означает для подчиненного устройства, что цикл закончен, поэтому устройство сбрасывает сигнал $\overline{\text{SSYN}}$, и все возвращается к первоначальному состоянию, когда все сигналы сброшены.

Стрелочки на временны́х диаграммах асинхронных шин (а иногда и синхронных шин) показывают причину и следствие какого-либо действия (см. рис. 3.36). Установка сигнала $\overline{\text{MSYN}}$ приводит к включению информационных линий, а также к установке сигнала $\overline{\text{SSYN}}$. Установка сигнала $\overline{\text{SSYN}}$, в свою очередь, вызывает отключение адресных линий, а также линий $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ и $\overline{\text{MSYN}}$.

Наконец, сброс сигнала $\overline{\text{MSYN}}$ вызывает сброс сигнала $\overline{\text{SSYN}}$, и на этом процесс считывания заканчивается.

Набор таких взаимообусловленных сигналов называется **полным квитированием**. Здесь, в сущности, наблюдается 4 события:

1. Установка сигнала $\overline{\text{MSYN}}$.
2. Установка сигнала $\overline{\text{SSYN}}$ в ответ на сигнал $\overline{\text{MSYN}}$.
3. Сброс сигнала $\overline{\text{MSYN}}$ в ответ на сигнал $\overline{\text{SSYN}}$.
4. Сброс сигнала $\overline{\text{SSYN}}$ в ответ на сброс сигнала $\overline{\text{MSYN}}$.

Разумеется, взаимообусловленность сигналов не является синхронной. Каждое событие вызывается предыдущим событием, а не импульсами генератора. Если какая-то пара устройств (задающее и подчиненное) работает медленно, это никак не влияет на другую пару устройств, которая может работать гораздо быстрее.

Преимущества асинхронной шины очевидны, хотя на самом деле большинство шин являются синхронными. Дело в том, что синхронную систему построить проще, чем асинхронную. Центральный процессор просто выдает сигналы, а память просто реагирует на них. Здесь нет никакой причинно-следственной связи, а если компоненты выбраны удачно, все работает и без квитирования. Кроме того, в разработку синхронных шин вложено очень много ресурсов.

Арбитраж шины

До этого момента мы неявно предполагали, что существует только одно задающее устройство шины — центральный процессор. В действительности микросхемы ввода-вывода могут становиться задающими устройствами при считывании информации из памяти и записи информации в память. Кроме того, они могут вызывать прерывания. Сопроцессоры также могут становиться задающими устройствами шины. Возникает вопрос: «Что происходит, когда задающим устройством шины становятся два или более устройств одновременно?» Чтобы предотвратить хаос, который может при этом возникнуть, нужен специальный механизм — так называемый **арбитраж шины**.

Арбитраж может быть централизованным или децентрализованным. Рассмотрим сначала централизованный арбитраж. Простой пример централизованного арбитража показан на рис. 3.37, *а*. В данном примере один арбитр шины определяет, чья очередь следующая. Часто механизм арбитража встраивается в микросхему процессора, но иногда используется отдельная микросхема. Шина содержит одну линию запроса (монтажное ИЛИ), которая может запускаться одним или несколькими устройствами в любое время. Арбитр не может определить, сколько устройств запрашивают шину. Он может определить только факт наличия или отсутствия запросов.

Когда арбитр обнаруживает запрос шины, он устанавливает линию предоставления шины. Эта линия последовательно связывает все устройства ввода-вывода (как в елочной гирлянде). Когда физически ближайшее к арбитру устройство получает сигнал предоставления шины, это устройство проверяет, нет ли запроса шины. Если запрос есть, устройство пользуется шиной, но не распространяет сигнал предоставления дальше по линии. Если запроса нет, устройство передает сигнал предоставления шины следующему устройству. Это устройство тоже про-

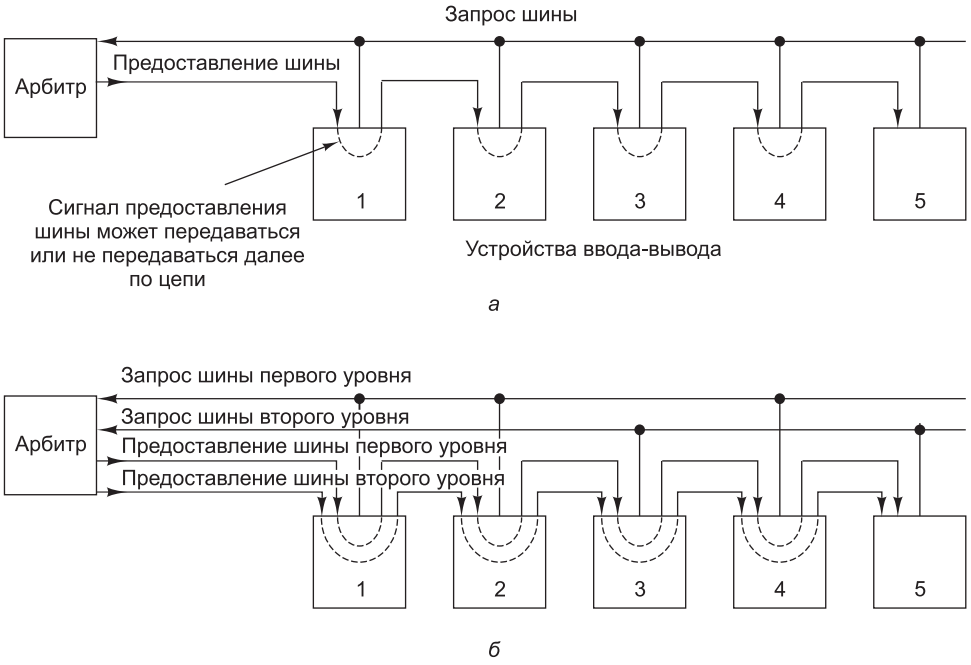


Рис. 3.37. Одноуровневый централизованный арбитраж шины с последовательным опросом (*а*); двухуровневый централизованный арбитраж (*б*)

веряет, есть ли запрос, и действует соответствующим образом в зависимости от наличия или отсутствия запроса. Передача сигнала предоставления шины продолжается до тех пор, пока какое-нибудь устройство не воспользуется предоставленной шиной. Такая система называется **системой последовательного опроса**. При этом приоритеты устройств зависят от того, насколько близко они находятся к арбитру. Ближайшее к арбитру устройство обладает наивысшим приоритетом.

Чтобы приоритеты устройств не зависели от расстояния от арбитра, в некоторых шинах поддерживается несколько уровней приоритета. На каждом уровне приоритета есть линия запроса шины и линия предоставления шины. На рис. 3.37, *б* изображено 2 уровня (хотя в действительности шины обычно поддерживают 4, 8 или 16 уровней). Каждое устройство связано с одним из уровней запроса шины, причем чем выше уровень приоритета, тем больше устройств привязано к этому уровню. На рис. 3.37, *б* можно видеть, что устройства 1, 2 и 4 обладают приоритетом уровня 1, а устройства 3 и 5 — приоритетом уровня 2.

Если одновременно запрашивается несколько уровней приоритета, арбитр предоставляет шину самому высокому уровню. Среди устройств одинакового приоритета реализуется система последовательного опроса. На рис. 3.37, *б* видно, что в случае конфликта устройство 2 «побеждает» устройство 4, а устройство 4 «побеждает» устройство 3. Устройство 5 имеет низший приоритет, поскольку оно находится в самом конце самого нижнего уровня.

Следует заметить, что с технической точки зрения линия предоставления шины уровня 2 не обязательно должна последовательно связывать устройства 1 и 2, поскольку они не могут посылать на нее запросы. Однако гораздо проще

провести все линии предоставления шины через все устройства, чем соединять устройства особым образом в зависимости от их приоритетов.

Некоторые арбитры содержат третью линию, которая устанавливается, как только устройство принимает сигнал предоставления шины, и получает шину в свое распоряжение. Как только эта линия подтверждения приема устанавливается, линии запроса и предоставления шины могут быть сброшены. В результате другие устройства могут запрашивать шину, пока первое устройство ее использует. К тому моменту, когда закончится текущая передача, следующее задающее устройство уже будет выбрано. Это устройство может начать работу, как только будет сброшена линия подтверждения приема. С этого момента начинается следующий цикл арбитража. Такая структура требует дополнительной линии и большего количества логических схем в каждом устройстве, но зато при этом циклы шины используются рациональнее.

В системах, где память связана с главной шиной, центральный процессор должен конкурировать со всеми устройствами ввода-вывода практически на каждом цикле шины. Чтобы решить эту проблему, можно предоставить центральному процессору самый низкий приоритет. При этом шина будет предоставляться процессору только в том случае, если она не нужна ни одному другому устройству. Центральный процессор всегда может подождать, а устройства ввода-вывода должны получить доступ к шине как можно быстрее, чтобы не потерять данные. Например, диски, вращающиеся с высокой скоростью, не могут ждать. Во многих современных компьютерах для решения этой проблемы память помещается на одну шину, а устройства ввода-вывода — на другую, поэтому им не приходится завершать работу, чтобы предоставить доступ к шине.

Возможен также децентрализованный арбитраж шины. Например, компьютер может содержать 16 приоритетных линий запроса шины. Когда устройству нужна шина, оно устанавливает свою линию запроса. Все устройства отслеживают все линии запроса, поэтому в конце каждого цикла шины каждое устройство может определить, обладает ли оно в данный момент наивысшим приоритетом и, следовательно, разрешено ли ей пользоваться шиной в следующем цикле. Такой метод требует большего количества линий, но зато избавляет от потенциальных затрат ресурсов на использование арбитра. В этом случае число устройств ограничивается числом линий запроса.

При другом типе децентрализованного арбитража используются только три линии независимо от того, сколько устройств имеется в наличии (рис. 3.38). Первая линия — монтажное ИЛИ. Она требуется для запроса шины. Вторая линия называется BUSY и означает занятость. Она запускается текущим задающим устройством шины. Третья линия служит для арбитража шины. Она

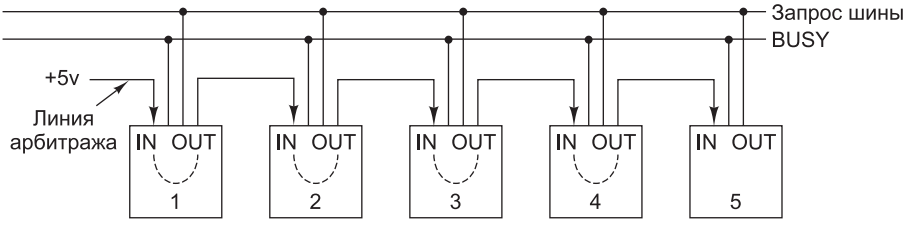


Рис. 3.38. Децентрализованный арбитраж шины

последовательно соединяет все устройства. Начало цепи связано с источником питания с напряжением 5 В.

Когда шина не требуется ни одному из устройств, линия арбитража передает сигнал всем устройствам. Чтобы получить доступ к шине, устройство сначала проверяет, свободна ли шина и установлен ли сигнал арбитража IN. Если сигнал IN не установлен, устройство не может стать задающим устройством шины. В этом случае оно сбрасывает сигнал OUT. Если сигнал IN установлен, устройство также сбрасывает сигнал OUT, в результате чего следующее устройство не получает сигнала IN и, в свою очередь, сбрасывает сигнал OUT. Следовательно, все следующие по цепи устройства не получают сигнал IN и сбрасывают сигнал OUT. В результате остается только одно устройство, у которого сигнал IN установлен, а сигнал OUT сброшен. Оно становится задающим устройством шины, устанавливает линию BUSY и сигнал OUT, после чего начинает передачу данных.

Немного поразмыслив, можно обнаружить, что из всех устройств, которым нужна шина, доступ к шине получает самое левое. Такая система напоминает систему последовательного опроса, только в данном случае нет арбитра, поэтому она стоит дешевле и работает быстрее. К тому же не возникает проблем со сбоями арбитра.

Принципы работы шины

До этого момента мы обсуждали только обычные циклы шины, когда задающее устройство (обычно центральный процессор) считывает информацию из подчиненного устройства (обычно из памяти) или записывает в него информацию. Однако существуют еще несколько типов циклов шины. Давайте рассмотрим некоторые из них.

Обычно за раз передается одно слово. При использовании кэш-памяти желательно сразу вызывать всю строку кэш-памяти (то есть 16 последовательных 64-разрядных слов). Однако часто передача блоками может быть более эффективна, чем такая последовательная передача информации. Когда начинается чтение блока, задающее устройство сообщает подчиненному устройству, сколько слов нужно передать (например, помещая общее число слов на информационные линии в период T_1). Вместо того чтобы выдать в ответ одно слово, задающее устройство выдает одно слово в течение каждого цикла до тех пор, пока не будет передано требуемое количество слов. На рис. 3.39 изображена такая же схема, как и на рис. 3.35, только с дополнительным сигналом $\overline{\text{BLOCK}}$, который указывает, что запрашивается передача блока. В данном примере считывание блока из четырех слов занимает 6 циклов вместо 12-ти.

Существуют также другие типы циклов шины. Например, если речь идет о системах с двумя или несколькими центральными процессорами на одной шине, нужно быть уверенным, что в конкретный момент только один центральный процессор может использовать определенную структуру данных в памяти. Чтобы упорядочить этот процесс, в памяти должна содержаться переменная, которая принимает значение 0, когда центральный процессор использует структуру данных, и 1, когда структура данных не используется. Если центральному процессору нужно получить доступ к структуре данных, он должен считать переменную, и если она равна 0, придать ей значение 1. Проблема заключается в том, что два

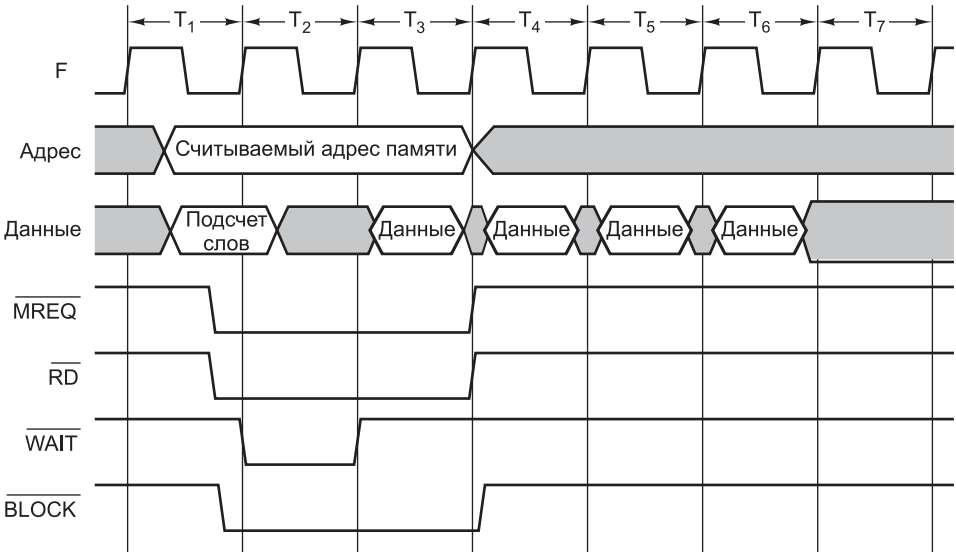


Рис. 3.39. Передача блока данных

центральных процессора могут считывать переменную на последовательных циклах шины. Если каждый процессор обнаружит, что переменная равна 0, а затем поменяет значение переменной на 1, как будто только он один использует эту структуру данных, то такая последовательность событий приведет к хаосу.

Чтобы не допустить подобную ситуацию, в мультипроцессорных системах предусмотрен специальный цикл шины, который дает возможность любому процессору считать слово из памяти, проверить и изменить его, а затем записать обратно в память; весь этот процесс происходит без освобождения шины. Такой цикл не дает возможности другим центральным процессорам использовать шину и, следовательно, мешать работе первого процессора.

Еще один важный цикл шины — цикл обработки прерываний. Когда центральный процессор командует устройству ввода-вывода произвести какое-то действие, он ожидает прерывания после завершения работы. Для сигнала прерывания нужна шина.

Поскольку может сложиться ситуация, когда несколько устройств одновременно захотят выполнить прерывание, здесь имеют место те же проблемы разрешения конфликтных ситуаций, что и в обычных циклах шины. Чтобы избежать таких проблем, нужно каждому устройству приписать определенный приоритет и для распределения приоритетов поддерживать централизованный арбитраж. Для этих целей существует стандартный, широко используемый интерфейс прерываний. В компьютерах IBM PC и последующих моделях для этого служит микросхема Intel 8259A. Она изображена на рис. 3.40.

До восьми контроллеров ввода-вывода могут быть непосредственно связаны с восемью входами IR*x* (Interrupt Request — запрос прерывания) микросхемы 8259A. Когда любое из этих устройств решит произвести прерывание, оно запускает свою линию входа. При активизации одного или нескольких входов контроллер 8259A выдает сигнал INT (INTerrupt — прерывание), который подается

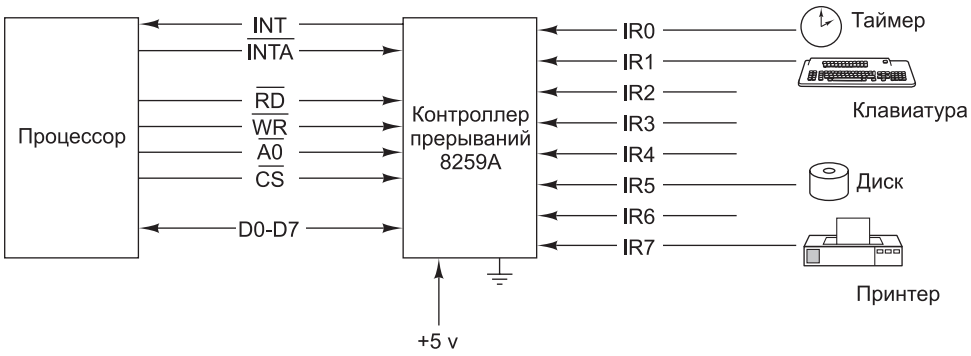


Рис. 3.40. Контроллер прерываний 8259A

на соответствующий вход центрального процессора. Если центральный процессор способен обработать прерывание, он посылает микросхеме 8259A импульс через вывод INTA (INTerrupt Acknowledge — подтверждение прерывания). В этот момент микросхема 8259A должна определить, на какой именно вход поступил сигнал прерывания. Для этого она помещает номер входа на информационную шину. Эта операция требует особого цикла шины. Центральный процессор использует этот номер для обращения к таблице указателей, которую называют таблицей **векторов прерываний**, чтобы найти адрес процедуры обработки этого прерывания.

Микросхема 8259A содержит несколько регистров, которые центральный процессор может считывать и записывать, используя обычные циклы шины и выводы $\overline{\text{RD}}$ (ReaD — чтение), $\overline{\text{WR}}$ (WRite — запись), $\overline{\text{CS}}$ (Chip Select — выбор элемента памяти) и $\overline{\text{A0}}$. Когда программное обеспечение обработало прерывание и готово получить следующее, оно записывает специальный код в один из регистров, который вызывает сброс сигнала INT микросхемой 8259A, если не появляется другое прерывание. Регистры также могут записываться для того, чтобы перевести микросхему 8259A в один из нескольких режимов, и для выполнения некоторых других функций.

При наличии более 8 устройств ввода-вывода, микросхемы 8259A могут соединяться каскадом. В самой экстремальной ситуации все 8 входов могут быть связаны с выходами еще 8 микросхем 8259A, соединяя до 64 устройств ввода-вывода в двухступенчатую систему обработки прерываний. Контроллер-концентратор ввода/вывода Intel ICH10 I/O, одна из микросхем чипсета Core i7, содержат два контроллера прерываний 8259A. Таким образом, ICH10 имеет 15 внешних прерываний — на 1 меньше 16 прерываний двух контроллеров 8259A, так как одно из прерываний используется для каскадного подключения второго контроллера 8259A. Микросхема 8259A содержит несколько выводов для каскадного соединения, но мы их опустили ради простоты. В наши дни 8259A является составной частью другой микросхемы.

Хотя приведенное описание никоим образом не исчерпывает всех вопросов разработки шин, оно дает достаточно информации для общего понимания принципов работы шины и принципов взаимодействия с шиной центрального процессора. Теперь мы перейдем от общего к частному и рассмотрим несколько конкретных примеров процессоров и их шин.

Примеры центральных процессоров

В этом разделе мы рассмотрим процессоры Intel Core i7, TI OMAP4430 и Atmel ATmega168 на уровне аппаратного обеспечения.

Intel Core i7

Core i7 — прямой потомок процессора 8088, который использовался в первой модели IBM PC. Презентация Core i7 состоялась в ноябре 2008 года. Публике было представлено четырехпроцессорное ЦПУ с 731 млн транзисторов, частотой до 3,2 ГГц и шириной строки 45 нанометра. Понятие «ширина строки» обозначает ширину проводников между транзисторами (и одновременно определяет размер самих транзисторов). Чем меньше эта величина, тем больше транзисторов умещается на одной микросхеме. По сути, закон Мура прогнозирует способность инженеров к дальнейшему уменьшению ширины строки. Помимо прочего, уменьшение этой величины позволяет повысить тактовую частоту. Для сравнения, диаметр человеческого волоса составляет 20–100 мкм (причем светлые волосы тоньше темных).

Исходный выпуск архитектуры Core i7 базировался на архитектуре «Nehalem», однако новые версии Core i7 строятся на базе более новой архитектуры «Sandy Bridge». Термином «архитектура» в этом контексте обозначается внутренняя организация центрального процессора, которой часто присваивается кодовое название. Обычно проектировщики компьютерных архитектур — люди серьезные, но иногда они придумывают для своих проектов очень остроумные кодовые названия. Например, архитектуры серии AMD K должны были разрушить позиции Intel на рынке процессоров для настольных систем, казавшиеся неуязвимыми. Для процессоров серии K было выбрано кодовое название «Kryptonite» — название единственного вещества, которое могло повредить Супермену, остроумный намекало на доминирование Intel.

Новая версия Core i7 на базе архитектуры «Sandy-Bridge» увеличилась до 1,16 млрд транзисторов. Она работает на скорости 3,5 ГГц с шириной строки 32 нанометра. Хотя Core i7 очень сильно отличается от процессора 8088 с его 29 000 транзисторов, он полностью совместим с 8088 и может выполнять двоичные программы, написанные для 8088 (не говоря уже о программах для всех процессоров, появившихся между Core i7 и 8088).

С точки зрения программного обеспечения Core i7 представляет собой 64-разрядную машину. Он поддерживает ту же стандартную промышленную архитектуру (ISA), что и процессоры 80386, 80486, Pentium, Pentium II, Pentium Pro, Pentium III и Pentium 4, включая те же регистры, те же команды и такую же встроенную систему обработки значений с плавающей точкой стандарта IEEE 754. Помимо этого, в Core i7 имеются новые команды, предназначенные в первую очередь для криптографических операций.

Core i7 является многоядерным процессором; таким образом, кремниевая подложка содержит несколько процессоров. Он продается с разным числом внутренних процессоров — от 2 до 6 (причем в ближайшем будущем их число должно увеличиться). Если программисты пишут параллельную программу с использованием потоков и блокировок, организация параллельного выполнения

на нескольких процессорах обеспечит существенный выигрыш по скорости. поддерживается технология **гиперпоточности**, позволяющая нескольким аппаратным потокам быть активными одновременно. Гиперпоточность позволяет осуществлять аппаратное переключение потоков во время очень коротких задержек (например, промахов кэша). Программное переключение потоков может происходить только во время очень длинных задержек (например, сбоев страниц), поскольку для его реализации требуются сотни тактов.

На уровне микроархитектуры Core i7 базируется на архитектуре своих предшественников Core 2 и Core 2 Duo. Процессор Core i7 может выполнять до четырех команд одновременно, что позволяет рассматривать его как 4-кратную суперскалярную машину. Микроархитектуру Core i7 мы обсудим в главе 4.

В процессорах Core i7 используется трехуровневый кэш. Каждый процессор Core i7 имеет 32-килобайтный кэш данных первого уровня (L1) и 32-килобайтный кэш команд первого уровня. У каждого ядра также имеется собственный 256-килобайтный кэш второго уровня (L2). Кэш второго уровня унифицирован, то есть позволяет хранить комбинацию команд и данных. Все ядра совместно используют один унифицированный кэш третьего уровня (L3), размер которого составляет от 4 до 15 Мбайт в зависимости от модели процессора. Трехуровневое кэширование значительно улучшает производительность процессора, но за счет возрастания стоимости кремниевых компонентов, так как у процессоров Core i7 общий объем кэша на одной подложке не может превышать 17 Мбайт.

Поскольку все микросхемы Core i7 содержат несколько процессоров с собственными кэшами данных, при изменении одним из процессоров слова, размещенного в его приватном кэше, могут возникать трудности. Если, предположим, другой процессор попытается считать это слово из памяти, он получит устаревшее значение, поскольку между изменением слова и его записью в память проходит некоторое время. В целях поддержания согласованности данных в памяти каждый ЦП в мультипроцессорной системе **следит** за шиной памяти на предмет поиска запросов на кэшированные слова. В случае обнаружения подобного рода запроса процессор предоставляет необходимые данные до того, как память передаст их другим потребителям. Технологию слежения мы рассмотрим в главе 8.

В системах с процессором Core i7 используются две внешние шины, обе они синхронные. Шина памяти DDR3 служит для доступа к главному динамическому ОЗУ; шина PCI Express — для взаимодействия с устройствами ввода-вывода. Высокопроизводительные версии Core i7 содержат несколько шин памяти и PCI Express, а также порт QPI (Quick Path Interconnect). Порт QPI связывает процессор с внешним мультипроцессорным соединением, что открывает возможность построения систем, в которых установлено более шести процессоров. Порт QPI отправляет и получает **запросы когерентности кэша**, а также другие управляющие сообщения для мультипроцессорных систем — например, межпроцессорные прерывания.

Основная проблема Core i7, как и у всех современных процессоров для настольных систем, заключается в объемах потребляемой мощности и выделяемого тепла. Чтобы избежать повреждения кремниевых компонентов, необходимо отводить тепло от процессора сразу же после его образования. Процессоры Core i7 в зависимости от частоты и модели потребляют от 17 до 150 Вт. Поэтому

Intel пребывает в постоянном поиске новых решений, которые позволили бы урегулировать проблему тепловыделения. Технологии охлаждения и теплопроводящие корпуса играют важную роль для защиты от выгорания кремниевых компонентов.

Микросхемы Core i7 поставляются в квадратном корпусе LGA с длиной стороны 37,5 мм. На нижней плоскости микросхемы находится 1155 площадок, из которых 286 используются для подачи питания, а 360 заземляются в целях шумоподавления. Площадки размещены в виде матрицы 40 × 40, причем ее центральный сегмент 17 × 25 не заполнен. Кроме того, по периметру асимметрично пропущены еще 20 площадок, за счет чего исключается возможность неправильной установки микросхемы в гнезде. Физическую компоновку Core i7 иллюстрирует рис. 3.41.

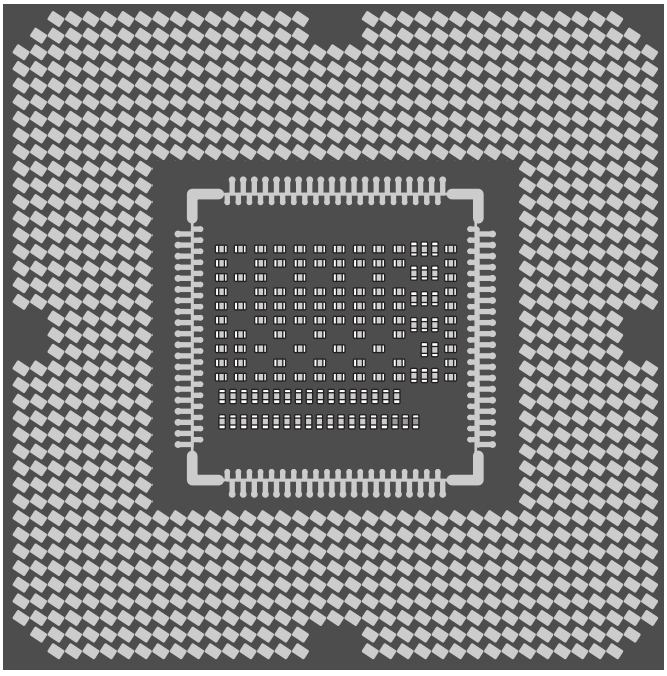


Рис. 3.41. Компоновка Core i7

Микросхема снабжена креплением для радиатора, который рассеивает тепло, и вентилятора, который охлаждает процессор. Чтобы получить некоторое представление о том, что собой представляет величина 150 Вт, поднесите руку к включенной электрической лампочке мощностью 150 Вт (только не дотрагивайтесь до нее). Вот такое количество тепла нужно рассеивать постоянно. Соответственно, когда Core i7 потеряет свои рабочие характеристики как процессор, он вполне сгодится в качестве нагревателя.

В соответствии с законами физики все, что выделяет большое количество тепла, должно потреблять большое количество энергии. В случае с портативным компьютером, который работает от батареи с ограниченным зарядом, потребление большого количества энергии нежелательно. Чтобы решить эту проблему,

компания Intel нашла способ переводить центральный процессор в режим пониженного энергоснабжения (состояние «сна»), если он не выполняет никаких действий, и вообще отключать его (вводить в состояние «глубокого сна»), если есть вероятность, что он не будет выполнять никаких действий некоторое время. Всего предусмотрено пять различных состояний — от полной активности до глубокого сна. В промежуточных состояниях одни функции работают (например, функция слежения кэша, обработка прерываний), другие — отключаются. В состоянии глубокого сна значения кэш-памяти и регистров сохраняются, а тактовый генератор и все внутренние блоки отключаются. Выход из «глубокого сна» происходит по специальному аппаратному сигналу. Видит ли Core i7 сны во время «глубокого сна», науке пока не известно.

Цоколевка процессора Core i7

Из 1155 контактов Core i7 для сигналов используются 447, для питания (с различным напряжением) — 286, для «земли» — 360; еще 62 зарезервированы на будущее. Для некоторых логических сигналов требуются два и более выводов (например, для запроса адреса памяти), поэтому существует только 131 вариант сигналов. Цоколевка Core i7 в несколько упрощенном виде представлена на рис. 3.42. С левой стороны рисунка показано 5 основных групп сигналов шины памяти; с правой стороны расположены прочие сигналы.

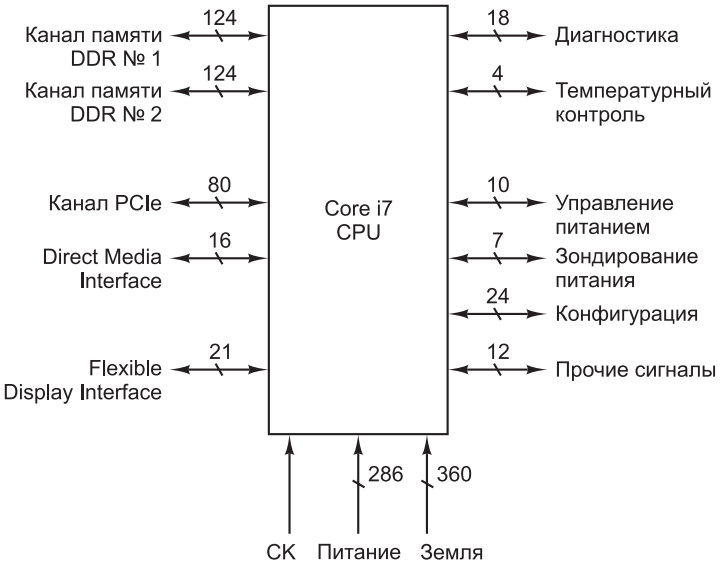


Рис. 3.42. Цоколевка процессора Core i7

Рассмотрим различные типы сигналов, начиная с сигналов шины. Первые два сигнала шины используются для взаимодействия с DDR3-совместимой динамической памятью. Группа сигналов предоставляет банку динамической памяти адрес, данные, управляющую информацию и синхронизацию. Core i7 поддерживает два независимых канала DDR3, работающих на частоте шины 666 МГц с передачей данных по фронту и по спаду сигнала; таким образом, возможно до

1333 млн взаимодействий в секунду. Интерфейс DDR3 является 64-разрядным, то есть два интерфейса DDR3 совместными усилиями ежесекундно предоставляют программам до 20 Гбайт данных.

Третья группа используется интерфейсом PCI, предназначенным для прямой связи периферийных устройств с центральным процессором Core i7. PCI Express — высокоскоростной последовательный интерфейс, в котором каждый последовательный канал образует «тракт» обмена данными с периферийными устройствами. Core i7 поддерживает интерфейс x16, то есть может одновременно использовать 16 трактов с совокупной пропускной способностью 16 Гбайт/с. Хотя канал является последовательным, через PCI Express передаются самые разнообразные команды, включая команды чтения с устройства, записи, прерывания и настройки конфигурации.

Следующая группа сигналов образует интерфейс **DMI** (Direct Media Interface), используемый для связи процессора Core i7 с комплектным чипсетом. Интерфейс DMI схож с PCI Express, хотя и работает на половине скорости последнего, поскольку четыре тракта могут обеспечить скорость передачи данных только до 2,5 Гбайт/с. Чипсет содержит полнофункциональную поддержку дополнительных периферийных интерфейсов, обычно необходимую для высокопроизводительных систем с многочисленными устройствами ввода-вывода. Чипсет Core i7 состоит из микросхем P67 и ICH10. Микросхема P67 обеспечивает поддержку интерфейсов SATA, USB, аудио, PCIe и флэш-памяти. Микросхема ICH10 обеспечивает поддержку наследных интерфейсов, включая интерфейс PCI и функциональность контроллера прерываний 8259A. Кроме того, ICH10 содержит много других схем: часы реального времени, таймеры событий и контроллеры прямого доступа к памяти (DMA). Существование таких микросхем значительно упрощает сборку полноценного персонального компьютера.

Core i7 может осуществлять прерывания тем же способом, что и 8088 (это требуется в целях совместимости), или использовать новую систему прерывания с устройством **APIC** (Advanced Programmable Interrupt Controller — **усовершенствованный программируемый контроллер прерываний**). Core i7 может действовать на любом из нескольких предустановленных напряжений, но процессор должен знать, на каком именно напряжении ему предстоит работать. Сигналы управления питанием используются для автоматического выбора напряжения источника питания, оповещения процессора о стабильности питания и ряда других родственных операций. С их же помощью осуществляется переход в различные состояния сна, которые, естественно, являются одними из инструментов управления питанием.

Несмотря на сложный механизм управления питанием, температура Core i7 может достигать очень высоких значений. Группа сигналов температурного контроля позволяет процессору оповещать окружающие устройства об опасности перегрева. Сюда относится, например, сигнал, который выдается центральным процессором, если его внутренняя температура превышает 130 °C (266 °F). Хотя если температура центрального процессора превышает 130 °C, он уже, вероятно, мечтает о выходе на пенсию и добросовестной службе в качестве нагревателя.

Впрочем, даже на таких запредельных температурах вам не придется беспокоиться о безопасности Core i7. Если внутренние датчики обнаруживают, что процессор вскоре перегреется, они запускают **терморегуляцию** — механизм, бы-

стро снижающий выделение тепла за счет того, что процессор работает только на каждом *N*-м такте. Чем выше значение *N*, тем сильнее замедляется процессор и тем быстрее он остывает. Конечно, за терморегуляцию приходится расплачиваться снижением производительности системы. До изобретения терморегуляции в случае недостаточно эффективного охлаждения процессор перегорал. Доказательства этих «мрачных времен» температурного контроля можно найти на YouTube (поищите по строке «exploding CPU»). Видеоролик поддельный, но проблема настоящая.

Сигнал Группа сигналов тактовой частоты отвечает за определение частоты системной шины. Группа диагностических сигналов предназначена для тестирования и отладки систем согласно стандарту IEEE 1149.1 JTAG. Группа сигналов инициализации обслуживает загрузку (запуск) системы.

Сигнал *CK* используется процессором для генерирования различных тактовых импульсов с частотой, кратной или дробной по отношению к частоте системного генератора. Для этого применяется устройство, называемое **системой автоподстройки по задержке**, или **DLL** (Delay-Locked Loop)

Группа диагностических сигналов предназначена для тестирования и отладки систем согласно стандарту IEEE 1149.1 JTAG (Joint Test Action Group). Наконец, в группу «прочих сигналов» отнесены разнородные сигналы, используемые для разных специальных целей.

Конвейерный режим шины памяти процессора Core i7

Современные процессоры, такие как Core i7, предъявляют жесткие требования к динамической памяти. Они работают гораздо быстрее, чем медленная динамическая память может выдавать значения, причем эта проблема усугубляется, когда несколько процессов выдают одновременные запросы. Чтобы процессор не простаивал, необходима максимально возможная производительность памяти. По этой причине шина памяти процессора Core i7 DDR3 работает в конвейерном режиме, когда в шине происходят одновременно до четырех операций. Понятие конвейера мы рассматривали в главе 2, когда говорили о конвейерных процессорах (см. рис. 2.4), но память тоже может быть конвейерной.

Чтобы конвейерный режим стал возможным, запросы к памяти Core i7 состоят из трех этапов:

1. Фаза активизации (ACT) памяти «открывает» строку динамической памяти, делая ее готовой для последующих обращений.
2. В фазе чтения (READ) или записи (WRITE) могут происходить обращения к отдельным словам открытой строки динамической памяти или к последовательным словам текущей строки динамической памяти с использованием пакетного режима.
3. Фаза предзаряда (PCHRG) «закрывает» текущую строку динамической памяти и готовит память к следующей команде активизации.

Конвейерная работа с памятью процессора Core i7 основана на том, что динамическая память DDR3 на микросхеме состоит из нескольких банков. **Банк** представляет собой блок динамической памяти, к которому процессор может обращаться параллельно с другими банками, даже находящимися на той же микросхеме. Типичная микросхема динамической памяти DDR3 содержит до восьми банков. Впрочем, спецификация интерфейса DDR3 разрешает не более

четырех параллельных обращений для одного канала DDR3. Временна́я диаграмма на рис. 3.43 показывает, как Core i7 выдает 3 обращения к трем разным банкам динамической памяти. Обращения полностью перекрываются, так что операции чтения на микросхеме динамической памяти выполняются параллельно. Связь между командами и последующими операциями на временно́й диаграмме обозначается стрелками.

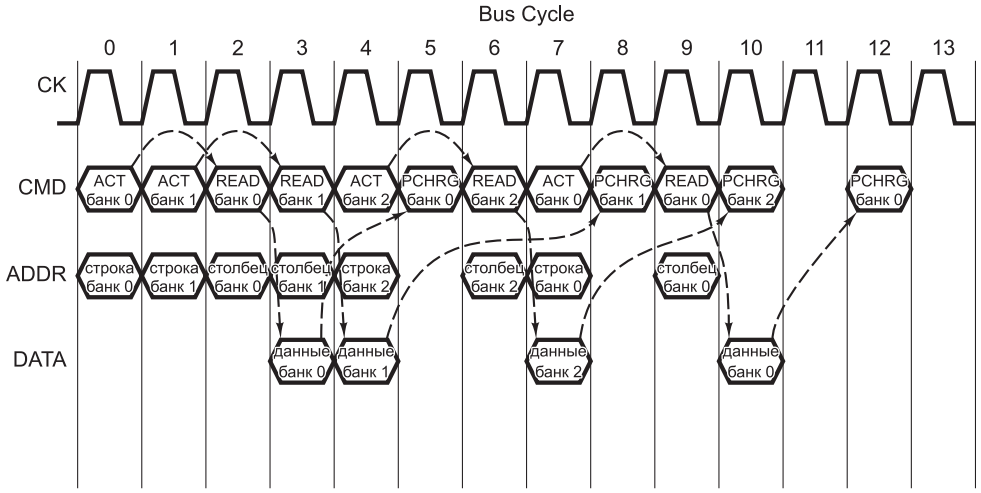


Рис. 3.43. Конвейерные обращения к памяти через интерфейс DDR3 процессора Core i7

Как видно из рис. 3.43, интерфейс памяти DDR3 имеет четыре основных сигнальных канала: синхронизация шины (CK), команда шины (CMD), адрес (ADDR) и данные (DATA). Сигнал синхронизации шины CK управляет всей работой шины. Командный сигнал CMD указывает, какая операция запрашивается у динамической памяти. Команда ACT задает адрес строки динамической памяти, открытой сигналом ADDR. При выполнении команды READ адрес столбца динамической памяти задается с использованием сигналов ADDR, а динамическая память выдает прочитанное значение спустя фиксированное время через сигналы DATA.

Наконец, команда PCHRG указывает банк, к которому применяется операция предзаряда, через сигналы ADDR. В нашем примере команда ACT должна предшествовать первой команде READ для того же банка на два цикла шины DDR3, а данные выдаются через один цикл после команды READ. Кроме того, операция PCHRG должна произойти по крайней мере на два цикла позже последней операции READ с тем же банком динамической памяти.

Параллелизм запросов памяти проявляется в перекрытии запросов READ к разным банкам динамической памяти. Первые два обращения READ к банкам 0 и 1 полностью перекрываются, производя результаты в циклах шины 3 и 4 соответственно. Обращение к банку 2 частично перекрывается с первым обращением к банку 1, и наконец, второе чтение из банка 0 частично перекрывается с обращением к банку 2.

Как Core i7 узнает, когда следует ожидать возвращения данных команды READ и когда можно выдавать новый запрос к памяти? Для этого он осущест-

вляет полное моделирование внутренней деятельности каждой подключенной микросхемы DDR3. Соответственно он ожидает возвращения данных в правильно выбранном цикле и знает, что операцию предзаряда не следует начинать раньше чем через два цикла после последней операции чтения. Core i7 может прогнозировать все эти события, потому что интерфейс памяти DDR3 работает синхронно, так что все операции занимают четко определенное количество тактов шины DDR3. Даже при наличии всей этой информации построение высокопроизводительного, полностью конвейерного интерфейса памяти DDR3 — нетривиальная задача, требующая применения многочисленных внутренних таймеров и детекторов конфликтов для реализации эффективной обработки запросов.

Однокристальная система Texas Instruments OMAP4430

В качестве второго примера процессора возьмем однокристальную систему Texas Instruments (TI) OMAP4430. OMAP4430 реализует набор команд ARM, а основной областью его применения являются мобильные и встроенные системы — смартфоны, планшетные компьютеры, интернет-гаджеты. Однокристальная система включает широкий диапазон таких устройств, чтобы при объединении ее с физической периферией (сенсорным экраном, флэш-памятью) формировалось полноценное компьютерное устройство.

OMAP4430 включает два ядра ARM A9, дополнительные ускорители и многочисленные интерфейсы периферийных устройств. Внутренняя организация OMAP4430 изображена на рис. 3.44. Ядра ARM A9 относятся к суперскалярной микроархитектуре ширины 2. Также на подложке OMAP4430 размещаются еще три процессора-ускорителя: графический процессор POWERVR SGX540, процессор обработки изображений (ISP, Image Signal Processor) и мультимедийный ускоритель IVA3.SGX540 — обеспечивает эффективную программируемую 3D-визуализацию и может рассматриваться как аналог графических процессоров для настольных компьютеров (хотя и уступающий им по скорости и мощности). ISP — программируемый процессор для эффективной обработки изображений (операции, необходимые в мощных цифровых фотокамерах). IVA3 реализует эффективное кодирование и декодирование видео с производительностью, достаточной для поддержки 3D-приложений (как, например, в ручных игровых устройствах). Также однокристальная система OMAP4430 содержит широкий спектр периферийных интерфейсов, включая сенсорные экраны и контроллеры клавиатуры, интерфейсы динамической и флэш-памяти, USB и HDMI. Фирма Texas Instruments опубликовала планы развития серии процессоров OMAP. В будущих архитектурах будет больше всего — больше ядер ARM, графических процессоров и разнообразных периферийных устройств.

Однокристальная система OMAP4430 впервые появилась в начале 2011 года. Она имела два ядра ARM A9, работавших на частоте 1 ГГц, с применением 45-нанометровой реализации. Ключевая особенность процессора OMAP4430 заключается в том, что он выполняет значительный объем вычислений с очень низкими энергозатратами, поскольку ориентируется на мобильные устройства, получающие питание от батарей. Чем эффективнее работает архитектура мобильного устройства, тем реже придется пользователю ставить устройство на зарядку.

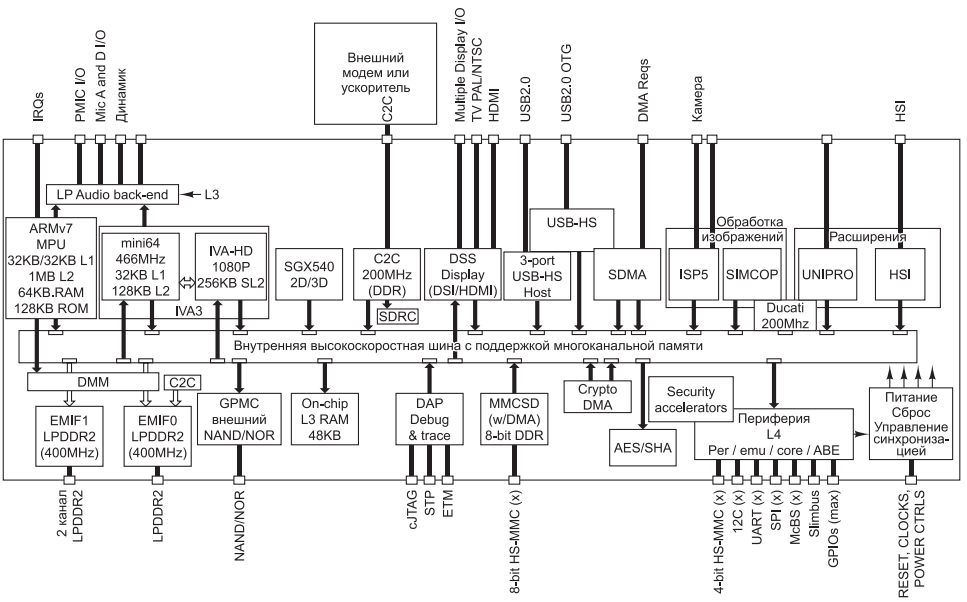


Рис. 3.44. Внутреннее строение однокристальной системы OMAP4430

Процессоры OMAP4430 выбраны с расчетом на достижение основной цели низкого энергопотребления. Графический процессор, ISP и IVA3 представляют собой программируемые ускорители, эффективно выполняющие вычисления при существенно меньших энергозатратах по сравнению с проведением тех же вычислений непосредственно на процессорах ARM A9. При полном энергопотреблении OMAP4430 использует всего 600 мВт мощности. Это составляет примерно 1/250 от энергопотребления высокопроизводительного Core i7. OMAP4430 также реализует очень эффективный спящий режим; при «засыпании» всех компонентов энергопотребление составляет всего 100 микроватт. Эффективные спящие режимы очень важны для мобильных устройств, проводящих много времени в режиме ожидания (как, например, сотовые телефоны). Чем меньше энергии расходуется в спящем режиме, тем дольше сотовый телефон сможет пребывать в ожидании.

Для дальнейшего сокращения энергопотребления в архитектуру OMAP4430 включены различные средства управления питанием, включая **динамическое масштабирование напряжения** и **ограничение питания**. Динамическое масштабирование напряжения позволяет компонентам медленнее работать на пониженном напряжении, что существенно снижает требования к питанию. Если вам не нужно, чтобы процессор выполнял вычисления на максимальной скорости, напряжение можно понизить — процессор работает медленнее, экономя значительное количество энергии. Механизм ограничения питания (power gating) использует еще более активный принцип управления питанием: неиспользуемый компонент полностью отключается и не потребляет энергии. Например, когда пользователь не смотрит видео на планшетном компьютере, видеопроцессор IVA3 полностью отключается и не потребляет энергии. И наоборот, во время просмотра IVA3 интенсивно выполняет работу по декодированию видеопотока, а два ядра ARM A9 «засыпают».

Несмотря на свою специализацию (экономия энергопотребления), ядра ARM A9 используют достаточно мощную микроархитектуру. За каждый такт они могут декодировать и выполнять до двух команд. Как мы узнаем в главе 4, эта скорость выполнения обеспечивает максимальную производительность микроархитектуры. Но не стоит полагать, что за каждый такт будут выполняться именно столько команд. Скорее, это гарантированная фирмой-изготовителем максимальная производительность; уровень, который не будет превышен процессором ни при каких условиях. Во многих тактах будет выполняться менее двух команд; это связано с наличием «препятствий», замедляющих выполнение команд и снижающих общую производительность системы. Для устранения таких ограничителей в ARM A9 встроена мощная система прогнозирования переходов, планировщик команд с изменением последовательности, и высокооптимизированная система памяти.

Система памяти OMAP4430 содержит два внутренних кэша L1 для каждого из процессоров ARM A9; 32-килобайтный кэш команд и 32-килобайтный кэш данных. Кэши обслуживаются двойными каналами с низким энергопотреблением LPDDR2. Стандарт LPDDR2 является производным от стандартного интерфейса памяти DDR2, но использует меньше проводников и работает на напряжениях, более эффективных по энергопотреблению. Кроме того, контроллер памяти содержит ряд оптимизаций (таких, как мозаичная предварительная выборка и поддержка ротации).

Хотя кэширование будет подробно рассмотрено в главе 4, сейчас о нем тоже стоит сказать пару слов. Вся основная память делится на строки кэша (блоки), состоящие из 32 байт. В кэше первого уровня хранятся 1024 наиболее интенсивно используемых строк команд и 1024 наиболее интенсивно используемых строк данных. Строки кэша, которые интенсивно используются, но не помещаются в кэше первого уровня, хранятся в кэше второго уровня. Этот кэш содержит как строки данных, так и строки команд от обоих процессоров ARM A9, смешанные произвольным образом. Кэш второго уровня содержит 32 768 строк основной памяти, к которым относились последние обращения. При промахе кэша первого уровня процессор отправляет идентификатор искомой строки кэшу второго уровня. Ответ содержит информацию, по которой процессор может определить, хранится ли указанная строка в кэше второго уровня, и если хранится — в каком состоянии. Если строка находится в кэше, процессор получает ее. Получение данных из кэша второго уровня выполняется за 19 тактов. Ожидание получается довольно долгим, поэтому квалифицированные программисты оптимизируют свою программу, чтобы она использовала меньше данных; тем самым повышается вероятность нахождения данных в быстром кэше первого уровня.

Если строка кэша отсутствует в кэше второго уровня, она загружается из основной памяти через интерфейс памяти LPDDR2. Интерфейс LPDDR2 в OMAP4430 встроен в кристалл, что делает возможным прямую связь OMAP4430 с динамической памятью. Для обращения к памяти процессор сначала отправляет микросхеме динамической памяти верхнюю часть адреса по 13 адресным линиям. Эта операция, называемая «активизацией» (ACTIVATE), загружает всю строку памяти из динамической памяти в буфер строк. Соответственно процессор может выдать несколько команд READ или WRITE, передавая остаток адреса по тем же 13 адресным линиям и отправляя (или получая) данные для операции по 32 линиям данных.

Во время ожидания результатов процессор может заниматься другой работой. Например, кэш-промах во время предварительной выборки команды не препятствует выполнению одной или нескольких уже выбранных команд, в каждой из которых могут быть задействованы данные, не находящиеся ни в одном кэше. Таким образом, даже у одного процессора может оставаться несколько незавершенных транзакций по двум интерфейсам LPDDR2. Контроллер памяти должен отслеживать текущие события и выдавать запросы к памяти в наиболее эффективном порядке.

Данные из памяти могут поступать частями по 4 байта. Операция с памятью может использовать чтение или запись в пакетном режиме, в котором происходит чтение или запись нескольких смежных адресов одной строки динамической памяти. Такой режим особенно эффективен для чтения или записи блоков кэша. Приведенное выше описание OMAP4430 (как и предшествующее ему описание Core i7) сильно упрощено, но суть происходящего в нем отражена.

Микросхема OMAP4430 содержит 547 выводов в корпусе **BGA** (рис. 3.45). Корпус BGA (Ball Grid Array) похож на LGA, но вместо квадратных площадок, используемых в LGA, в нем используются маленькие металлические шарики. Эти два типа корпусов несовместимы, что лишний раз доказывает справедливость поговорки «круглую дырку не заткнешь квадратной пробкой». Выводы OMAP4430 образуют прямоугольную матрицу размером 28 × 26, в которой отсутствуют два внутренних кольца шариков, а также две асимметричных полустроки и полустолбца для исключения возможности неправильной установки микросхемы в гнезде BGA.

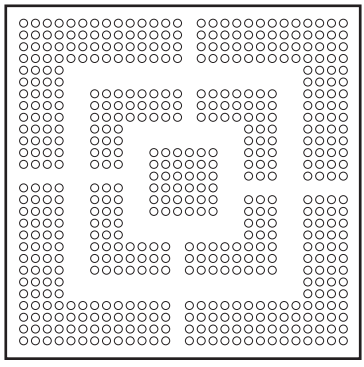


Рис. 3.45. Микросхема однокристальной системы OMAP4430

Трудно сравнивать CISC-микросхему (такую, как Core i7) с RISC-микросхемой (такой, как OMAP4430) на основании одной лишь тактовой частоты. Например, у двух ядер ARM A9 пиковая скорость выполнения достигает четырех команд на такт; в этом OMAP4430 почти сравнивается со суперскалярными процессорами ширины 4 у Core i7. Однако Core i7 быстрее выполняет программы, потому что он содержит до шести процессоров, тактовая частота которых в 3,5 раза выше (3,5 ГГц), чем у OMAP4430. Может показаться, что OMAP4430 напоминает черепаху, которая пытается обогнать зайца Core i7, однако «черепаха» расходует намного меньше энергии и первой придет к финишу — особенно если заряд батареи у «зайца» не слишком велик.

Микроконтроллер Atmel ATmega168

Core i7 и OMAP4430 — высокопроизводительные процессоры, разработанные для создания быстродействующих вычислительных устройств (Core i7 предназначен для настольных компьютеров, OMAP4430 — для мобильных систем). Однако существуют и другие компьютеры, на самом деле куда более многочисленные — так называемые встроенные системы. В этом разделе мы познакомимся с ними.

Не будет преувеличением сказать, что практически любое электронное устройство стоимостью более 100 долларов содержит встроенный компьютер. Телевизоры, сотовые телефоны, электронные секретари, микроволновые печи, видеокамеры, видеомагнитофоны, лазерные принтеры, системы охранной сигнализации, слуховые аппараты, электронные игры и многие другие устройства (их можно перечислять до бесконечности) управляются компьютером. При этом упор делается не на высокую производительность, а на низкую стоимость встроенного компьютера, что приводит к несколько другому соотношению достоинств и недостатков по сравнению с процессорами, которые мы обсуждали до сих пор.

В главе 1 мы уже упоминали о том, что в настоящее время наиболее распространенным микроконтроллером является ATmega168. Такая популярность, в первую очередь, обусловлена его низкой стоимостью. Как вы вскоре убедитесь, ATmega168 — это универсальная микросхема, к которой очень легко и недорого подключать другие устройства. Ее физическая компоновка показана на рис. 3.46.

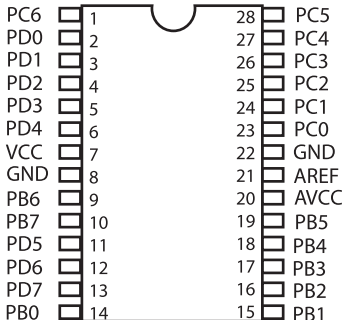


Рис. 3.46. Физическая компоновка микросхемы ATmega168

Как видно из рисунка, ATmega168 обычно поставляется в стандартном корпусе с 28 выводами (хотя для отдельных вариантов применения предусмотрены и другие корпуса). Сразу же бросается в глаза некоторая странность этой микросхемы по сравнению с двумя предыдущими примерами: у нее нет адресных линий и линий данных. Дело в том, что микросхема не предназначена для подключения к памяти — только к устройствам. Вся память (статическая и флэш-память) содержится в самом процессоре, вследствие чего необходимость в отдельных адресных линиях и линиях данных отпадает (рис. 3.47).

Вместо адресных линий и линий данных ATmega168 содержит 27 портов цифрового ввода-вывода, 8 линий портов B и D и 7 линий порта C. Линии цифрового ввода-вывода предназначены для подключения периферийных устройств ввода-вывода, причем программа инициализации может задать режим работы каждой линии (ввод или вывод). Например, при использовании в микроволновой печи одна цифровая линия может получать ввод от датчика «открытой

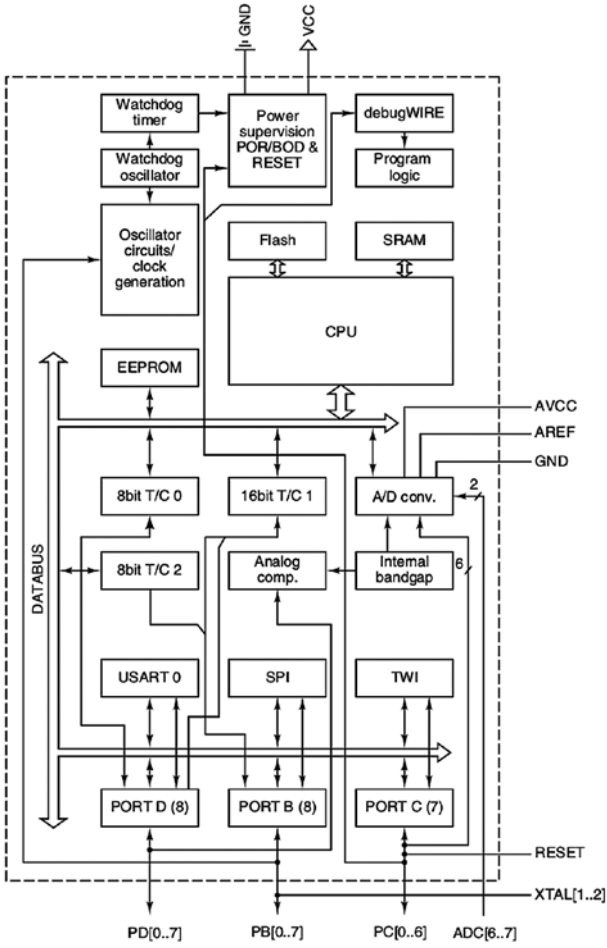


Рис. 3.47. Внутренняя архитектура и логическая компоновка ATmega168

двери», а другая — выдавать выходной сигнал для включения и выключения СВЧ-генератора. Перед включением СВЧ-генератора программное обеспечение ATmega168 проверяет, закрыта ли дверь печи. Если дверь внезапно откроется, программа должна срочно отключить питание. Впрочем, на практике также всегда применяется аппаратная блокировка.

Кроме того, шесть линий порта C могут быть настроены для аналогового ввода-вывода. Линии аналогового ввода-вывода могут читать уровень входного напряжения и задавать уровень выходного напряжения. Так, в контексте предыдущего примера у некоторых микроволновых печей имеется датчик, который позволяет пользователю разогреть еду до заданной температуры. Датчик температуры подключается к входу порта C, программа читает напряжение датчика и преобразует его в температуру с использованием функции преобразования для конкретного датчика. Остальные линии ATmega168 — вход питания (VCC), два заземления (GND) и две линии для настройки схем аналогового ввода-вывода (AREF, AVCC).

С точки зрения внутренней архитектуры ATmega168, как и OMAP4430, представляет собой однокристальную систему с широким набором внутренних устройств и памяти. ATmega168 оснащается 16 Кбайт внутренней флэш-памяти для хранения редко изменяющейся энергонезависимой информации (например, команд программы). Также имеется 1 Кбайт EEPROM — энергонезависимой памяти, запись в которую может осуществляться на программном уровне. В EEPROM хранятся данные конфигурации системы. Возвращаясь к нашему примеру с микроволновой печью: скажем, в EEPROM будет храниться бит, определяющий формат отображения времени (12- или 24-часовой). ATmega168 также содержит до 1 Кбайт внутренней статической памяти, в которой программное обеспечение может хранить временные переменные.

Во внутреннем процессоре используется набор команд AVR. Он состоит из 131 команды, каждая из которых имеет длину 16 бит. Сам процессор является 8-разрядным; это означает, что он работает с 8-разрядными значениями данных, а размер его внутренних регистров составляет 8 бит. В набор команд входят специальные команды, позволяющие 8-разрядному процессору эффективно работать с большими типами данных. Например, для выполнения 16-разрядного сложения (или сложения с большей разрядностью) процессор поддерживает команду «сложения с переносом», которая суммирует два значения и перенос от предыдущей операции. Среди остальных внутренних компонентов — часы реального времени и разнообразная интерфейсная логика: поддержка последовательных каналов, поддержка каналов широтно-импульсной модуляции, канала I2C (шина Inter-IC), контроллеры аналогового и цифрового ввода-вывода.

Примеры шин

Шины соединяют компьютерную систему в единое целое. В этом разделе мы рассмотрим популярные шины PCI и USB. Шина PCI — основная шина периферийного ввода-вывода, используемая в современных PC. Она существует в двух разновидностях: более старая шина PCI и новая, более скоростная шина PCI Express (PCIe). USB — чрезвычайно популярная шина ввода-вывода для периферийных устройств с невысоким быстродействием (таких, как мыши и клавиатуры). Вторая и третья версии USB работают на более высоких скоростях. В следующих разделах мы поочередно рассмотрим каждую из этих шин.

Шина PCI

В первых компьютерах IBM PC большинство приложений были текстовыми. С появлением Windows постепенно вошли в употребление графические пользовательские интерфейсы. Ни одно из этих приложений особо не нагружало шину ISA. Однако с течением времени появилось множество различных приложений, в том числе игр, для которых требовалось полноэкранное видео, и ситуация коренным образом изменилась.

Давайте произведем небольшие вычисления. Рассмотрим монитор размером 1024 × 768 с 3 байтами на пиксел. Одно экранное изображение содержит 2,25 Мбайт данных. Для воспроизведения плавных движений требуется 30 кадров в секунду, и, следовательно, скорость передачи данных должна быть

67,5 Мбайт/с. В действительности дело обстоит гораздо хуже, поскольку, чтобы передать изображение, данные нужно передать с жесткого диска, компакт-диска или DVD-диска через шину в память. Затем данные должны поступить в графический адаптер (тоже через шину). Таким образом, только для передачи видео пропускная способность шины должна быть 135 Мбайт/с, без учета потребностей центрального процессора и других устройств.

Максимальная частота передачи данных предшественницы PCI — шины ISA — составляла 8,33 МГц. Шина ISA способна передавать два байта за цикл, поэтому ее максимальная пропускная способность составляет 16,7 Мбайт/с. Шина EISA может передавать 4 байта за цикл. Ее пропускная способность достигает 33,3 Мбайт/с. Ясно, что ни одна из них совершенно не соответствует тому, что требуется для полноэкранного видео.

С современным видео формата Full HD дело обстоит еще хуже. Для него необходимо воспроизведение 1920 × 1080 с частотой 30 кадров в секунду, поэтому скорость передачи данных должна составлять 155 Мбайт/с (или 310 Мбайт/с, если данные должны проходить по шине дважды). Разумеется, шина EISA даже близко не соответствовала этим требованиям.

В 1990 году компания Intel разработала новую шину с гораздо более высокой пропускной способностью, чем у шины EISA. Эту шину назвали **PCI** (Peripheral Component Interconnect — **взаимодействие периферийных компонентов**). Компания Intel запатентовала шину PCI и сделала все патенты всеобщим достоянием, так что любая компания могла производить периферийные устройства для этой шины без каких-либо выплат за право пользования патентом. Компания Intel также сформировала промышленный консорциум PCI Special Interest Group, который должен был заниматься дальнейшими усовершенствованиями шины PCI. Все эти действия привели к тому, что шина PCI стала чрезвычайно популярной. Фактически в каждом компьютере Intel (начиная с Pentium), а также во многих других компьютерах есть шина PCI. Шина PCI подробно описана в литературе [Shanley and Anderson, 1999; Solari and Willse, 2004].

Первая шина PCI передавала 32 бита за цикл и работала на частоте 33 МГц (время цикла — 30 нс), общая пропускная способность составляла 133 Мбайт/с. В 1993 году появилась шина PCI 2.0, а в 1995 году — PCI 2.1. Шина PCI 2.2 подходит и для портативных компьютеров (где требуется экономии заряда батареи). В конце концов удалось получить шину PCI, которая работает на частоте 66 МГц, способна передавать 64 бита за цикл, а ее общая пропускная способность составляет 528 Мбайт/с. При такой производительности полноэкранное видео вполне достижимо (если диск и другие устройства системы справляются со своей частью работы). Во всяком случае, шина PCI не является «узким местом» системы.

Хотя 528 Мбайт/с — достаточно высокая скорость передачи данных, все же здесь есть некоторые проблемы. Во-первых, этого недостаточно для шины памяти. Во-вторых, шина PCI несовместима со всеми старыми платами ISA. По этой причине компания Intel решила разрабатывать компьютеры с тремя и более шинами, как показано на рис. 3.48. Здесь мы видим, что центральный процессор может обмениваться информацией с основной памятью через специальную шину памяти, а шину ISA можно связать с шиной PCI. Такая архитектура в 1990-х годах удовлетворяла всем современным на тот момент требованиям и поэтому использовалась в большинстве систем.

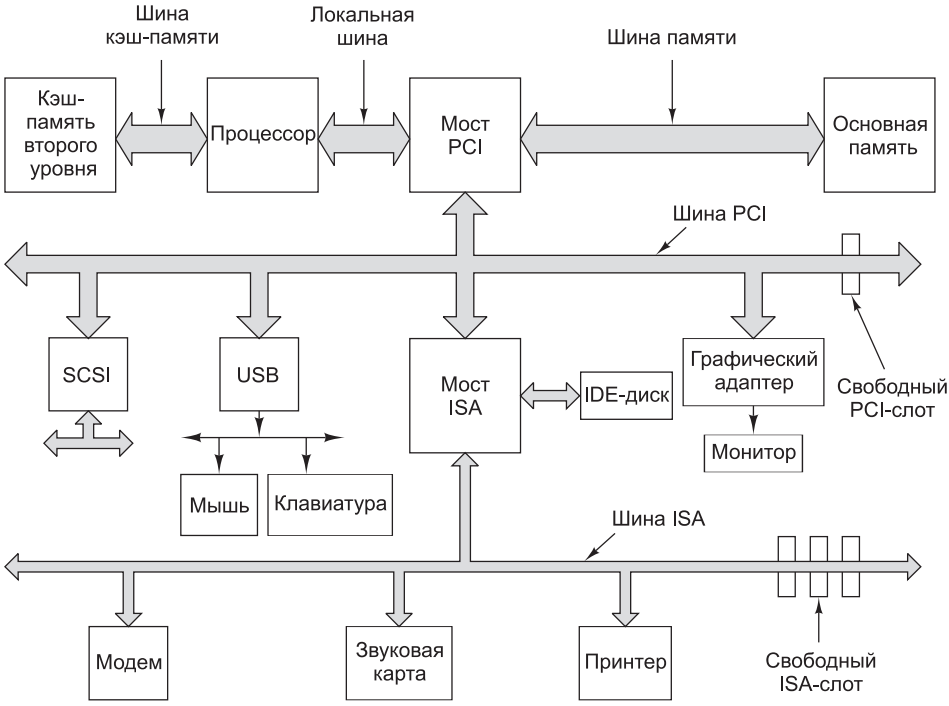


Рис. 3.48. Архитектура типичной системы первых поколений Pentium. (Толщина линий шины обозначает ее пропускную способность. Чем толще линия, тем выше пропускная способность)

Ключевыми компонентами данной архитектуры являются мосты между шинами (эти микросхемы выпускает компания Intel — отсюда такой интерес к проекту). Мост PCI связывает центральный процессор, память и шину PCI. Мост ISA связывает шину PCI с шиной ISA, а также поддерживает один или два IDE-диска. Практически все PC, использующие эту архитектуру, выпускаются с одним или несколькими свободными PCI-слотами для подключения дополнительных высокоскоростных периферийных устройств и с одним или несколькими ISA-слотами для подключения низкоскоростных периферийных устройств.

Преимущество системы, изображенной на рис. 3.48, состоит в том, что шина между центральным процессором и памятью имеет чрезвычайно высокую пропускную способность, шина PCI также обладает высокой пропускной способностью и хорошо подходит для взаимодействия с высокоскоростными периферийными устройствами (SCSI-дисками, графическими адаптерами и т. п.), и при этом еще могут использоваться старые платы ISA. На рисунке также изображена шина USB, которую мы будем обсуждать далее в этой главе.

Было бы неплохо, если бы существовал только один тип плат PCI. К сожалению, это не так. Платы отличаются потребляемой мощностью, разрядностью и синхронизацией. Старые компьютеры обычно используют напряжение 5 В, а новые — 3,3 В, поэтому шина PCI поддерживает то и другое. Коннекторы одни и те же (они отличаются только двумя небольшими пластмассовыми вставками, не позволяющими вставить плату на 5 В в шину PCI на 3,3 В, и наоборот). К счастью,

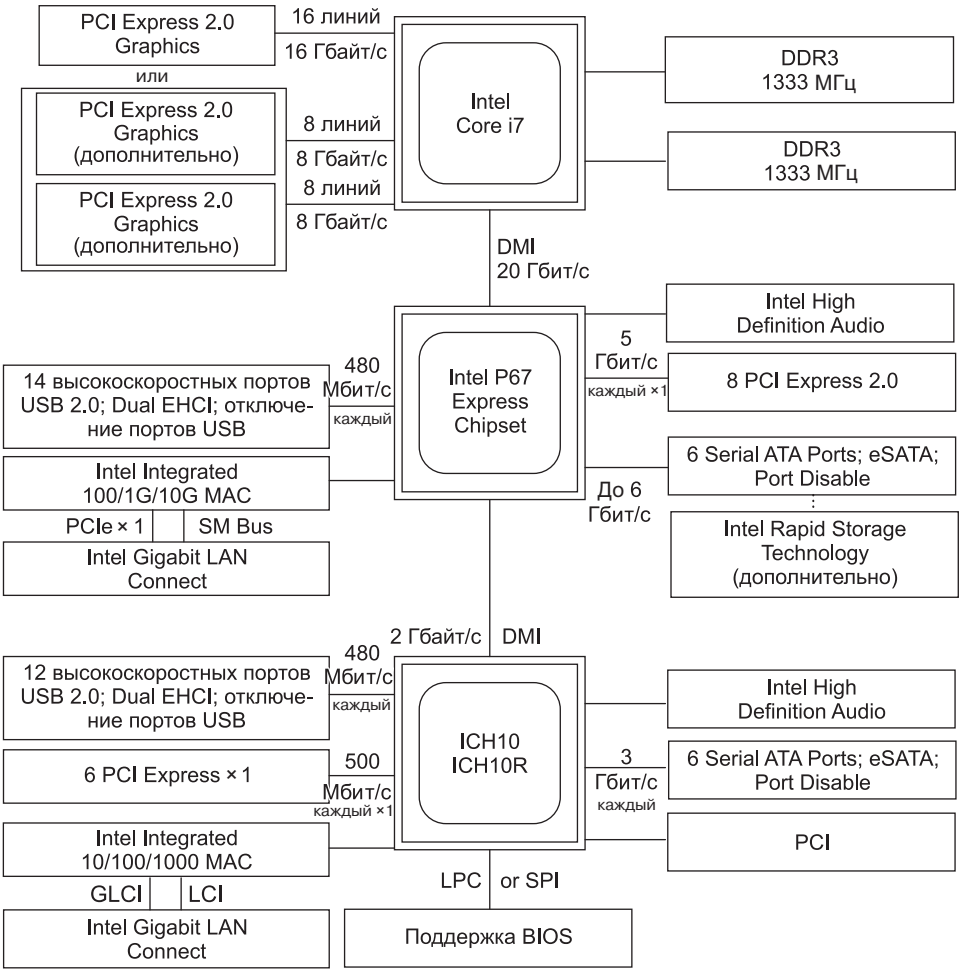


Рис. 3.49. Структура шин в современной системе Core i7

существуют и универсальные платы, которые поддерживают оба напряжения и которые можно вставить в любой слот. Платы отличаются не только напряжением, но и разрядностью. Существует два типа плат: 32-разрядные и 64-разрядные. 32-разрядные платы содержат 120 выводов; 64-разрядные платы содержат те же 120 выводов плюс 64 дополнительных вывода. Шина PCI, поддерживающая 64-разрядные платы, может поддерживать и 32-разрядные, но обратное не верно. Наконец, шины PCI и соответствующие платы могут работать на частоте либо 33 МГц, либо 66 МГц. В обоих случаях контакты идентичны. Отличие состоит в том, что один из выводов связывается либо с источником питания, либо с землей.

К концу 1990-х годов шина ISA была окончательно похоронена участниками рынка, и в новых системах ее поддержка уже не предусматривалась. В связи с тем, что разрешение экрана постоянно увеличивалось (достигнув величины 1600 × 1200 точек), равно как и спрос на полноэкранное видео со стандартной частотой кадров, особо актуальное в интерактивных играх, компания Intel разработала новую шину,

предназначенную исключительно для обмена данными с графическим адаптером. Эта шина называется **AGP** (Accelerated Graphics Port — **ускоренный графический порт**). Ее первая версия, AGP 1.0, работала на скорости 264 Мбайт/с, и эта величина была принята за 1x. Недостаток скорости (по сравнению с PCI) компенсировался узкой специализацией на управлении графическим адаптером. Впоследствии были разработаны новые версии шины — в частности, AGP 3.0 работает на скорости 2,1 Гбайт/с (8x). Сегодня даже высокопроизводительная шина AGP 3.0 уступает более быстрым новинкам — прежде всего, PCI Express, способной передавать данные со скоростью до 16 Гбайт/с по высокоскоростному последовательному каналу. Схема современной системы на базе Core i7 показана на рис. 3.49.

В современной системе на базе Core i7 ряд интерфейсов встраивается прямо в микросхему процессора. Два канала памяти DDR3, работающих со скоростью 1333 транзакции/с, соединяются с основной памятью и обеспечивают суммарную пропускную способность 10 Гбайт/с на канал. Также в процессор интегрируется канал PCI Express на 16 линий, который может быть настроен для работы в режиме одной 16-разрядной шины PCI Express или двух независимых 8-разрядных шин PCI Express. 16 линий совместно обеспечивают для устройств ввода-вывода пропускную способность в 16 Гбайт/с.

Процессор соединен с основной мостовой микросхемой P67 через последовательный интерфейс DMI (Direct Media Interface) со скоростью 20 Гбит/с (2,5 Гбайт/с). P67 предоставляет интерфейс к нескольким современным высокопроизводительным интерфейсам ввода-вывода. Поддерживаются 8 дополнительных линий PCI Express и дисковые интерфейсы SATA. P67 также реализует 14 интерфейсов USB 2.0, 10G Ethernet и аудиоинтерфейс.

Микросхема ICH10 обеспечивает поддержку интерфейсов старых устройств. Она соединяется с P67 через медленный интерфейс DMI. ICH10 реализует шину PCI, 1G Ethernet, порты USB ports и старые версии PCI Express и SATA. В новых системах ICH10 микросхема может отсутствовать; она необходима только в том случае, если система должна поддерживать старые интерфейсы.

Работа шины PCI

Шины PCI являются синхронными, как и все шины PC, восходящие к первой модели IBM PC. Все транзакции в шине PCI осуществляются между **задающим** и **подчиненным** устройствами. Чтобы не увеличивать число выводов на плате, адресные и информационные линии объединяются. При этом достаточно 64-х выводов для всей совокупности адресных и информационных сигналов, даже если PCI работает с 64-разрядными адресами и 64-разрядными данными.

Объединенные адресные и информационные выводы функционируют следующим образом. При считывании во время первого цикла задающее устройство передает адрес на шину. Во время второго цикла задающее устройство удаляет адрес, и шина переключается таким образом, чтобы подчиненное устройство могло ее использовать. Во время третьего цикла подчиненное устройство выдает запрашиваемые данные. При записи шине не нужно переключаться, поскольку задающее устройство передает в нее и адрес, и данные. Тем не менее минимальная транзакция занимает три цикла. Если подчиненное устройство не может дать ответ в течение трех циклов, то вводится режим ожидания. Допускаются пересылки блоков неограниченного размера, а также некоторые другие типы циклов шины.

Арбитраж шины PCI

Чтобы использовать шину PCI, устройство сначала должно получить к ней доступ. Шина PCI управляется централизованным арбитром, как показано на рис. 3.50. В большинстве случаев арбитр шины встраивается в один из мостов между шинами. От каждого PCI-устройства к арбитру тянутся две специальные линии. Одна из них (REQ#) используется для запроса шины, а вторая (GNT#) — для получения разрешения на доступ к шине. (Примечание. REQ# — обозначение $\overline{\text{REQ}}$ в терминологии PCI.)

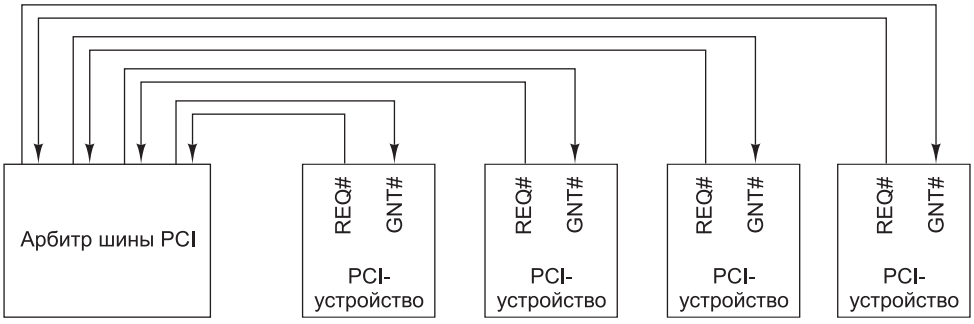


Рис. 3.50. У шины PCI имеется централизованный арбитр

Чтобы сделать запрос на доступ к шине, PCI-устройство (в том числе и центральный процессор) устанавливает сигнал REQ# и ждет, пока арбитр не установит сигнал GNT#. Если арбитр установил сигнал GNT#, то устройство может использовать шину в следующем цикле. Алгоритм, которым руководствуется арбитр, не определяется в спецификации шины PCI. Допустимы циклический арбитраж, приоритетный арбитраж, а также другие схемы арбитража. Разумеется, хороший арбитр должен быть справедлив, чтобы не заставлять отдельные устройства ждать целую вечность.

Шина предоставляется для одной транзакции, хотя продолжительность этой транзакции теоретически не ограничена. Если устройству нужно совершить вторую транзакцию и ни одно другое устройство не запрашивает шину, оно может занять шину снова, хотя обычно между транзакциями требуется вставлять пустой цикл. Однако при особых обстоятельствах (при отсутствии конкуренции на доступ к шине) устройство может совершать последовательные транзакции без пустых циклов между ними. Если задающее устройство выполняет очень длительную передачу, а какое-нибудь другое устройство выдало запрос на доступ к шине, арбитр может сбросить сигнал на линии GNT#. Предполагается, что задающее устройство следит за линией GNT#, и при сбросе сигнала устройство должно освободить шину в следующем цикле. Такая система позволяет осуществлять очень длинные передачи (что весьма рационально) при отсутствии конкуренции на доступ к шине, однако при этом она быстро реагирует на запросы шины, поступающие от других устройств.

Сигналы шины PCI

Шина PCI поддерживает ряд обязательных (табл. 3.4) и ряд дополнительных сигналов (табл. 3.5). Оставшиеся выводы используются для питания, земли

и разнообразных сопутствующих сигналов. В столбцах «Задающее устройство» и «Подчиненное устройство» указывается, какое из устройств устанавливает сигнал при обычной транзакции. Если сигнал устанавливается другим устройством (например, CLK), оба столбца остаются пустыми.

Таблица 3.4. Обязательные сигналы шины PCI

Сигнал	Количество линий	Задающее устройство	Подчиненное устройство	Комментарий
CLK	1			Тактовый генератор (33 МГц или 66 МГц)
AD	32	Да	Да	Объединенные адресные и информационные линии
PAR	1	Да		Бит четности для адреса или данных
C/BE#	4	Да		Во-первых, команда шине, во-вторых, битовый массив, который показывает, какие байты из слова нужно считать (или записать)
FRAME#	1	Да		Указывает, что установлены сигналы AD и C/BE
IRDY#	1	Да		При чтении означает, что задающее устройство готово принять данные; при записи — что данные находятся в шине
IDSEL	1	Да		Считывание конфигурационного пространства
DEVSEL#	1		Да	Подчиненное устройство распознало свой адрес и ждет сигнала
TRDY#	1		Да	При чтении означает, что данные находятся на линиях AD; при записи — что подчиненное устройство готово принять данные
STOP#	1		Да	Подчиненное устройство требует немедленно прервать текущую транзакцию
PERR#	1			Обнаружена ошибка четности данных
SERR#	1			Обнаружена ошибка четности адреса или системная ошибка
REQ#	1			Арбитраж шины — запрос на доступ к шине
GNT#	1			Арбитраж шины — предоставление шины
RST#	1			Перезагрузка системы и всех устройств

Таблица 3.5. Дополнительные сигналы шины PCI

Сигнал	Количество линий	Задающее устройство	Подчиненное устройство	Комментарий
REQ64#	1	Да		Запрос на осуществление 64-разрядной транзакции
ACK64#	1		Да	Разрешение 64-разрядной транзакции
AD	32	Да		Дополнительные 32 бита адреса или данных
PAR64	1	Да		Проверка четности для дополнительных 32 бит адреса или данных
C/BE#	4	Да		Дополнительные 4 бита для указания, какие байты из слова нужно считать (или записать)
LOCK	1	Да		В многопроцессорных системах блокировка шины при осуществлении транзакции одним из процессоров
SBO#	1			В многопроцессорных системах обращение к кэш-памяти другого процессора
SDONE	1			В многопроцессорных системах сигнал о завершении слежения
INTx	4			Запрос прерывания
JTAG	5			Сигналы тестирования IEEE 1149.1 JTAG
M66EN	1			Сигнал связывается с источником питания или с землей (66 МГц или 33 МГц)

Теперь давайте рассмотрим каждый сигнал шины PCI отдельно. Начнем с обязательных (32-разрядных) сигналов, а затем перейдем к дополнительным (64-разрядным). Сигнал CLK управляет шиной. Большинство сигналов синхронизируется с ним. У шины PCI транзакция начинается на спаде сигнала CLK, то есть не в начале цикла, а в середине.

Сигналы AD (их 32) нужны для адресов и данных (для передач по 32 бита). Обычно адрес устанавливается во время первого цикла, а данные — во время третьего. Сигнал PAR — это бит четности для сигнала AD. Сигнал C/BE# выполняет две функции. Во время первого цикла он содержит команду (считать одно слово, считать блок и т. п.). Во время второго цикла он содержит массив размером 4 бита, который показывает, какие байты 32-разрядного слова действительны. Используя сигнал C/BE#, можно считать 1, 2 или 3 байта из слова, а также все слово целиком.

Сигнал FRAME# устанавливается задающим устройством, чтобы начать транзакцию. Этот сигнал сообщает подчиненному устройству, что адрес и команды

в данный момент действительны. При чтении одновременно с сигналом FRAME# устанавливается сигнал IRDY#. Он сообщает, что задающее устройство готово принять данные. При записи сигнал IRDY# устанавливается позже, когда данные уже переданы в шину.

Сигнал IDSEL связан с тем, что у каждого устройства PCI должно быть конфигурационное пространство на 256 байт, которое другие устройства могут считывать (установив сигнал IDSEL). Это конфигурационное пространство содержит характеристики устройства. В некоторых операционных системах механизм автоматического конфигурирования (Plug and Play, PnP) использует это пространство, чтобы выяснить, какие устройства подключены к шине.

А теперь рассмотрим сигналы, которые устанавливаются подчиненным устройством. Сигнал DEVSEL# означает, что подчиненное устройство распознало свой адрес на линиях AD и готово участвовать в транзакции. Если сигнал DEVSEL# не поступает в течение определенного промежутка времени, задающее устройство предполагает, что подчиненное устройство, к которому направлено обращение, либо отсутствует, либо неисправно.

Следующий сигнал — TRDY#. Его подчиненное устройство устанавливает при чтении, чтобы сообщить, что данные находятся на линиях AD, и при записи, чтобы сообщить, что оно готово принять данные.

Следующие три сигнала используются для передачи сообщений об ошибках. Один из них, сигнал STOP#, устанавливается подчиненным устройством, если произошла какая-нибудь неполадка и нужно прервать текущую транзакцию. Следующий сигнал, PERR#, используется для сообщения об ошибке четности в данных на предыдущем цикле. Для чтения этот сигнал устанавливается задающим устройством, для записи — подчиненным устройством. Необходимые действия должно предпринимать устройство, получившее этот сигнал. Наконец, сигнал SERR# служит для сообщения об адресных и системных ошибках.

Сигналы REQ# и GNT# предназначены для арбитража шины. Они устанавливаются не тем устройством, которое является задающим в данный момент, а тем, которое желает стать задающим. Последний обязательный сигнал, RST#, применяется для перезагрузки системы, которая происходит либо при нажатии пользователем кнопки RESET, либо если какое-нибудь системное устройство обнаруживает фатальную ошибку. После установки этого сигнала компьютер перезагружается.

Перейдем к дополнительным сигналам, большинство из которых связано с расширением разрядности с 32 до 64 бит. Сигналы REQ64# и ACK 64# позволяют задающему устройству попросить разрешение осуществить 64-разрядную транзакцию, а подчиненному устройству принять эту транзакцию. Сигналы AD, PAR64 и C/BE# являются расширениями соответствующих 32-разрядных сигналов.

Следующие три сигнала не связаны с переходом с 32 на 64 бита. Они используются в многопроцессорных системах. Не все платы PCI поддерживают такие системы, поэтому эти сигналы отнесены к дополнительным. Сигнал LOCK позволяет блокировать шину для параллельных транзакций. Следующие два сигнала связаны с фазой слежения, позволяющей сохранить согласованность кэшей разных процессоров.

Сигналы INT*x* нужны для запроса прерываний. Плата PCI может содержать до четырех логических устройств, каждое из которых имеет собственную линию запроса прерываний. Сигналы JTAG предназначены для процедуры тестирования

IEEE 1149.1 JTAG. Наконец, сигнал M66EN связывается либо с источником питания, либо с землей, что определяет тактовую частоту. Она не должна меняться во время работы системы.

Транзакции на шине PCI

Шина PCI в действительности очень проста (для современной шины, конечно). Чтобы лучше понять это, рассмотрим временну́ю диаграмму на рис. 3.51. Здесь мы видим транзакцию чтения, за ней следуют пустой цикл и транзакция записи, которая осуществляется тем же задающим устройством.

Во время цикла T_1 на спаде синхронизирующего сигнала задающее устройство помещает адрес на линии AD и команду на линии C/BE#. Затем задающее устройство устанавливает сигнал FRAME#, чтобы начать транзакцию.

Во время цикла T_2 задающее устройство переключает шину, чтобы подчиненное устройство могло воспользоваться ею во время цикла T_3. Задающее устройство также изменяет сигнал C/BE#, чтобы указать, какие байты в слове ему нужно считать.

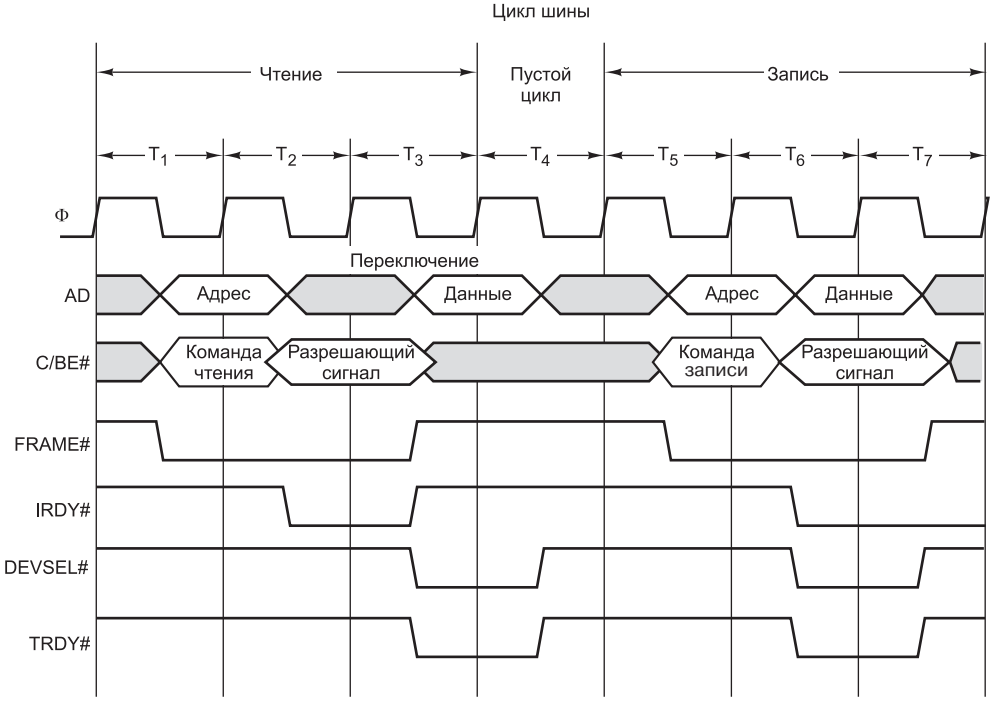


Рис. 3.51. Примеры 32-разрядных транзакций на шине PCI. Во время первых трех циклов происходит операция чтения, затем идет пустой цикл, а следующие три цикла — операция записи

Во время цикла T_3 подчиненное устройство устанавливает сигнал DEVSEL#. Этот сигнал сообщает задающему устройству, что подчиненное устройство получило адрес и собирается ответить. Подчиненное устройство также помещает данные на линии AD и выдает сигнал TRDY#, который сообщает задающему устройству о данном действии. Если подчиненное устройство не может ответить быстро, оно

не снимает сигнал DEVSEL#, извещающий о присутствии этого устройства, но при этом не устанавливает сигнал TRDY# до тех пор, пока не сможет передать данные. При такой процедуре вводится один или несколько периодов ожидания.

В нашем примере (а также часто на практике) следующий цикл — пустой. Мы видим, что в цикле T_5 то же самое задающее устройство инициирует процесс записи. Сначала оно как обычно помещает адрес и команду на шину. В следующем цикле оно выдает данные. Поскольку линиями AD управляет одно и то же устройство, цикл переключения не требуется. В цикле T_7 память принимает данные.

PCI Express

Возможностей шины PCI вполне достаточно для большинства современных приложений, однако потребность в ускорении ввода-вывода постепенно усложняет некогда стройную внутреннюю архитектуру ПК. Рисунок 3.49 наглядно свидетельствует о том, что шина PCI более не является центральным элементом, сводящим воедино компоненты ПК. Эту роль теперь исполняет мост.

Суть проблемы заключается в том, что со временем появляется все больше устройств ввода-вывода, слишком быстрых для шины PCI. Разгон тактовой частоты шины — далеко не лучшее решение, поскольку только усугубляет проблемы с расфазировкой шины, перекрестными помехами между проводниками и емкостным сопротивлением. При появлении каждого нового устройства, которое оказывается слишком быстрым для шины PCI (будь то графический адаптер, жесткий диск, сетевой контроллер и т. д.), разработчикам Intel приходится создавать очередной специализированный порт, с помощью которого мост позволяет этому устройству обходить шину PCI. Естественно, такое решение не рассчитано на долгосрочную перспективу.

Еще один недостаток шины PCI состоит в чрезмерных габаритах плат. Стандартные платы PCI обычно имеют размеры 17,5 × 10,7 см, а компактные платы — 12,5 × 3,6 см. Ни один из вариантов не умещается в корпусе современных портативных компьютеров, не говоря уже карманных моделях. В то же время производители постоянно уменьшают размеры выпускаемых устройств. Кроме того, некоторые производители планируют перейти на новую схему размещения устройств в корпусах ПК — а именно размещать процессор и память в отдельном закрытом отсеке, а жесткий диск — внутри монитора. С платами PCI такое решение не реализуемо.

Сейчас предлагается несколько вариантов решения указанных проблем, но, скорее всего, победителем в конкурентной борьбе окажется технология **PCI Express**, которую активно продвигает Intel. Несмотря на название, она не имеет почти ничего общего с шиной PCI; более того — это вообще не шина. Тем не менее маркетологи решили не избавляться от названия «PCI» — благо, оно у всех на слуху. Сейчас наличие этой шины у компьютера стало уже стандартом. Посмотрим, что она собой представляет.

Архитектура PCI Express

Суть технологии PCI Express заключается в замене параллельной шины с ее многообразием задающих и подчиненных устройств высокоскоростными двухточечными последовательными соединениями. Это решение знаменует собой

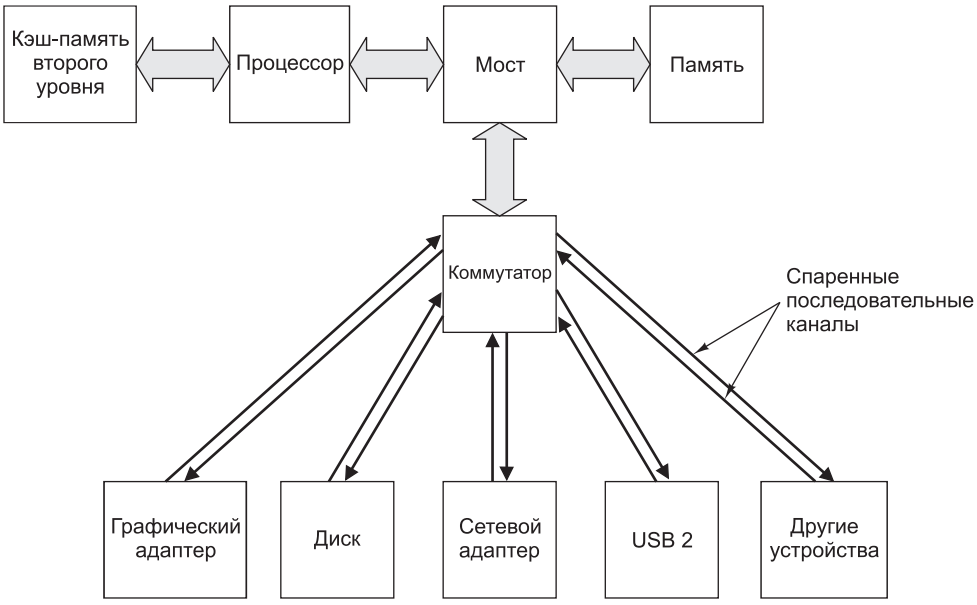


Рис. 3.52. Стандартная компоновка системы PCI Express

окончательный отход от шинной топологии, реализованной в шинах ISA/EISA/PCI, и переход на топологию локальных сетей (особенно коммутируемых сетей Ethernet). Основная идея такова: по сути, ПК — это набор микросхем процессора, памяти и устройств ввода-вывода, которые необходимо соединить между собой. Учитывая это обстоятельство, PCI Express исполняет роль универсального коммутатора, соединяющего микросхемы по последовательным каналам. Стандартная конфигурация PCI Express изображена на рис. 3.52.

Как видно из рисунка, процессор, память и кэш подключены к мосту традиционным способом. Новым элементом здесь является подключенный к мосту коммутатор (иногда он встраивается непосредственно в микросхему моста). Между каждой микросхемой устройства ввода-вывода, с одной стороны, и коммутатором, с другой, устанавливается двухточечное соединение. Любое такое соединение состоит из двух однонаправленных каналов — по одному в каждом из направлений между устройством и коммутатором. Каналы состоят из двух проводов (сигнального и заземляющего), что обеспечивает высокую помехозащищенность в ходе высокоскоростной передачи сигналов. Такая архитектура отличается от предыдущей большей унификацией и равноправием всех устройств.

Три основные момента отличают архитектуру PCI Express от архитектуры PCI. Два из них мы уже рассмотрели — это наличие централизованного коммутатора, пришедшего на смену принципу многоотводной шины, и применение узких последовательных двухточечных соединений вместо широкой параллельной шины. Третье отличие не столь очевидно. Концептуальная модель, на которой основана шина PCI, сводится к тому, что задающее устройство шины передает подчиненным устройствам команды на чтение слова или блока из нескольких слов. PCI Express основывается на другом принципе, предусматривающем от-

правку пакетов данных от одного устройства другому. Понятие **пакета**, состоящего из заголовка и полезной нагрузки, заимствовано из сетевых технологий. В **заголовке** содержится управляющая информация, а значит, отпадает потребность в многочисленных управляющих сигналах, которые играют важную роль при передаче по шине PCI. **Полезная нагрузка** содержит непосредственно передаваемые данные. Таким образом, ПК, поддерживающий технологию PCI Express, напоминает миниатюрную сеть с коммутацией пакетов.

Помимо вышеперечисленных наиболее важных изменений есть и менее заметные. В частности, в пакетах использован более надежный по сравнению с PCI код обнаружения ошибок. Далее, физическая длина соединения между микросхемой и коммутатором увеличилась до 50 см, за счет чего стало удобнее менять положение компонентов системы. Поскольку к базовому коммутатору можно подключить другой коммутатор, сформировав, таким образом, древовидную структуру, повышается степень расширяемости системы. Кроме того, устройства в рамках PCI Express поддерживают «горячее» подключение, а значит, их можно снимать и монтировать непосредственно в процессе работы. Наконец, так как последовательные коннекторы значительно меньше старых PCI-коннекторов, ничто не мешает производителям разрабатывать компактные устройства и компьютеры. Таким образом, PCI Express решительно отходит от принципов работы шины PCI.

Стек протоколов PCI Express

Следуя модели сети с коммутацией пакетов, технология PCI Express реализуется на основе многоуровневого стека протоколов. **Протоколом** называется набор правил, определяющих механизм взаимодействия между двумя сторонами. Соответственно, стек протоколов — это иерархическая система протоколов, которые регламентируют различные аспекты взаимодействия на тех или иных уровнях. Рассмотрим для примера деловое письмо. Существуют определенные правила относительно местоположения и содержания шапки письма, адреса получателя, даты, формы приветствия, тела письма, подписей и т. д. Все эти условности можно обобщенно назвать протоколом делового письма. Помимо этого, есть стандарты, касающиеся размера и формата конверта, размещения штампа и тому подобных тонкостей. Эти два уровня и соответствующие протоколы независимы друг от друга. К примеру, можно полностью изменить формат письма, положив его в стандартный конверт, и наоборот. Подобные многоуровневые протоколы, которые делают возможной модульную разработку с высоким уровнем гибкости, уже несколько десятилетий широко применяются в области сетевого ПО. В технологии PCI Express сделана попытка реализовать их в аппаратном обеспечении «шины».

Стек протоколов PCI Express изображен на рис. 3.53, *а*.

Рассмотрим уровни по восходящей. Самым нижним является **физический уровень**. Он отвечает за передачу битов от отправителя к получателю по двухточечному соединению. Каждое такое соединение состоит из одной или нескольких пар симплексных (однонаправленных) каналов. В простейшем случае на каждое направление выделяется по одной паре, но также допустимо наличие 2, 4, 8, 16 или 32 пар. Каналы, входящие в пары, называются **полосами** (lanes). На каждое направление должно быть выделено одинаковое количество полос. В первом поколении предусматривается скорость передачи данных от 2,5 Гбайт/с для каждого направления, но через некоторое время эта цифра, вероятно, дойдет до 10 Гбайт/с.

Рис. 3.53. Стек протоколов PCI Express (*а*); формат пакета (*б*)

В отличие от шин ISA, EISA и PCI, в технологии PCI Express не предусмотрен тактовый генератор. Устройства вправе начинать передачу в любой момент, как только им будет, что передавать. Такая свобода, с одной стороны, повышает быстродействие, с другой порождает проблему. Предположим, что 1 кодируется напряжением +3 В, а 0 — напряжением 0 В. Если первые несколько байтов равны нулю, как получатель узнает о том, что ему передаются данные? Действительно — последовательность нулевых битов трудно отличить от простоя канала. Эта проблема решается при помощи так называемой **8/10-разрядной кодировки**. Согласно этой схеме, 1 байт фактических данных кодируется при помощи 10-разрядного символа. Из 1024 возможных 10-разрядных символов выбираются такие, которые за счет достаточного количества фронтов без задающего генератора обеспечивают синхронизацию отправителя и получателя по границам битов. В силу применения 8/10-разрядной кодировки суммарная пропускная способность канала, равная 2,5 Гбайт/с, сужается до фактической пропускной способности 2 Гбайт/с.

Канальный уровень отвечает за передачу пакетов. На этом уровне к заголовку и полезной нагрузке, переданным с уровня транзакций, добавляется порядковый номер и код исправления ошибок — так называемый **CRC** (Cyclic Redundancy Check — **циклический контроль избыточности**). CRC-код генерируется путем применения определенного алгоритма к заголовку и полезной нагрузке. По получении пакета устройство проводит аналогичные вычисления с заголовком и данными и сравнивает результат с величиной, указанной в пакете. Если два результата совпадают, первоначальному отправителю отсылается **пакет подтверждения** правильности полученных данных. В противном случае получатель делает запрос на повторную передачу. Таким образом, значительно повышаются показатели целостности данных по сравнению с шиной PCI, в которой не реализованы средства контроля и повторной передачи данных.

Во избежание перегрузки медленного получателя пакетами, исходящими от быстрого отправителя, реализуется механизм **управления потоками**. Этот механизм основывается на выдаче получателем отправителю определенного количества разрешений на передачу пакетов — в зависимости от объема свободного пространства, необходимого для их хранения. Исчерпав ранее выданные разрешения, отправитель должен приостановить передачу и дождаться новых разрешений. Такая схема, распространенная во всех сетях, предотвращает потерю данных вследствие несовпадения скоростей отправителя и получателя.

На **уровне транзакций** выполняются все операции шины. К примеру, для считывания слова из памяти нужно выполнить две транзакции, из которых одну инициирует процессор или канал DMA, запрашивающий данные, другую — целевой объект (поставщик данных). Впрочем, чтение и запись — не единственные

операции, которые выполняются на уровне транзакций. Этот уровень, в частности, расширяет возможности передачи пакетов, предоставляемые канальным уровнем. Каждая полоса на уровне каналов подразделяется на несколько (числом до восьми) **виртуальных каналов**, по каждому из которых передаются данные того или иного типа. На уровне транзакций пакеты маркируются согласно классу трафика, определяющему ряд свойств, таких как «высокий приоритет», «низкий приоритет», «запрет слежения», «допускается доставка вне последовательности» и т. д. Выстраивая порядок обработки пакетов, коммутатор, помимо прочего, основывается на информации из маркеров.

Любая транзакция проходит в одном из четырех адресных пространств:

- пространство памяти (при выполнении стандартных операций чтения и записи);
- пространство ввода-вывода (для адресации регистров устройств);
- конфигурационное пространство (для инициализации системы и т. д.);
- пространство сообщений (для отправки сигналов, прерываний и т. д.).

Пространства памяти и ввода-вывода аналогичны традиционным — тем, что реализованы в современных системах. В конфигурационном пространстве возможна реализация разного рода механизмов, например автоматического конфигурирования (PnP). Пространство сообщений принимает на себя функции многочисленных ныне управляющих сигналов. Обойтись без этого пространства нельзя, ведь в PCI Express отсутствуют предусмотренные в шине PCI линии управления.

Программный уровень выступает посредником между PCI Express и операционной системой. Помимо прочего, на нем предусмотрен режим эмуляции шины PCI, позволяющий устанавливать в компьютерах, оснащенных PCI Express, старые операционные системы без каких-либо изменений. Естественно, при работе в таких условиях реализация всех возможностей PCI Express неосуществима, однако обеспечение обратной совместимости является необходимой мерой — по крайней мере, до того момента, пока во всех операционных системах не будет полностью реализована поддержка PCI Express. Опыт показывает, что этот процесс займет немало времени.

Информационный поток, характерный для PCI Express, показан на рис. 3.53, *б*. Команда, поступающая на программный уровень, передается на уровень транзакций, где из нее формируются заголовок и полезная нагрузка. Затем эти компоненты отправляются на канальный уровень, на котором в заголовке пакета устанавливается порядковый номер, а в хвостовике — CRC-код. Далее расширенный пакет передается на физический уровень, где с обоих концов к нему добавляются параметры кадра, и получившийся в результате физический пакет передается от отправителя получателю. На стороне получателя происходит обратный процесс — заголовок и хвостовик кадра канального уровня удаляются, а результат передается на уровень транзакций.

Концепция присоединения дополнительных данных на каждом последующем уровне стека протоколов применяется в компьютерных сетях уже очень долго и успешно. Основное отличие между сетевыми технологиями и PCI Express заключается в том, что в первом случае код, действующий на различных уровнях стека, почти всегда является программным и управляется операционной системой. В PCI Express, напротив, операции на всех уровнях реализуются аппаратно.

Структура PCI Express довольно сложна. Ее подробное описание имеется в работах [Mayhewand Krishnan, 2003; Solari and Congdon, 2005]. К тому же технология продолжает развиваться — в 2007 году была выпущена версия PCIe 2.0. Она поддерживает скорость передачи 500 Мбайт/с на канал и до 32 каналов, так что общая пропускная способность достигает 16 Гбайт/с. В 2011 году вышла версия PCIe 3.0, в которой кодировка изменилась с 8/10-разрядной на 128/130-разрядную, с возможностью выполнения до 8 миллиардов транзакций в секунду — вдвое больше, чем у PCIe 2.0.

Шина USB

Шины PCI и PCI Express очень хорошо подходят для соединения высокоскоростных периферийных устройств, но использовать интерфейс PCI для низкоскоростных устройств ввода-вывода (например, мыши и клавиатуры) было бы неэффективно. Изначально каждое стандартное устройство ввода-вывода соединялось с компьютером особым образом, при этом для добавления новых устройств использовались свободные ISA- и PCI-слоты. К сожалению, такая схема имеет некоторые недостатки.

Например, каждое новое устройство ввода-вывода часто оснащается собственной платой ISA или PCI. Пользователь при этом должен сам установить переключатели и перемычки на плате и удостовериться, что настроенная плата не конфликтует с другими платами. Затем пользователь должен открыть системный блок, аккуратно вставить плату, закрыть системный блок и включить компьютер. Для многих этот процесс очень сложен и часто приводит к ошибкам. Кроме того, число ISA- и PCI-слотов очень мало (обычно два или три). Автоматически конфигурируемые (PnP) платы исключают необходимость установки переключателей, но пользователь все равно должен открывать компьютер и вставлять туда плату. К тому же количество слотов шины ограничено.

В 1993 году представители семи компаний (Compaq, DEC, IBM, Intel, Microsoft, NEC и Nothern Telecom) собрались вместе, чтобы разработать шину, оптимально подходящую для подсоединения низкоскоростных устройств. Потом к ним примкнули сотни других компаний. Результатом их работы стала шина **USB** (Universal Serial Bus — **универсальная последовательная шина)**, которая сейчас широко используется в персональных компьютерах. Более подробное описание USB приведено в литературе [Anderson, 1997; Tan 1997].

Некоторые требования, изначально составившие основу проекта:

- пользователи не должны устанавливать переключатели и перемычки на платах и устройствах;
- пользователи не должны открывать компьютер, чтобы установить новые устройства ввода-вывода;
- должен существовать только один тип кабеля, подходящий для соединения всех устройств;
- устройства ввода-вывода должны получать питание через кабель;
- должна быть возможность подсоединения к одному компьютеру до 127 устройств;
- система должна поддерживать устройства реального времени (например, звуковые устройства, телефон);

- должна быть возможность устанавливать устройства во время работы компьютера;
- установка нового устройства не должна требовать перезагрузки компьютера;
- производство новой шины и устройств ввода-вывода для нее не должны требовать больших затрат.

Шина USB удовлетворяет всем этим условиям. Она разработана для низкоскоростных устройств (клавиатур, мышей, фотоаппаратов, сканеров, цифровых телефонов и т. д.). Общая пропускная способность первой версии шины (USB 1.0) составляла 1,5 Мбит/с. Версия 1.1 работает на скорости 12 Мбит/с, что вполне достаточно для принтеров, цифровых камер и многих других устройств. Версия 2.0 поддерживает устройства со скоростью до 480 Мбит/с, достаточной для поддержки внешних дисков, веб-камер высокого разрешения и сетевых интерфейсов. В недавно появившейся версии USB 3.0 скорость возросла до 5 Гбит/с; только время покажет, какие новые и требовательные приложения породит этот сверхскоростной интерфейс.

Шина USB состоит из **корневого хаба** (root hub), который вставляется в разъем главной шины (см. рис. 3.49). Этот корневой хаб (часто называемый корневым концентратором) содержит разъемы для кабелей, которые могут подсоединяться к устройствам ввода-вывода или к дополнительным хабам, чтобы увеличить количество разъемов. Таким образом, топология шины USB представляет собой дерево с корнем в корневом хабе, который находится внутри компьютера. Коннекторы кабеля со стороны устройства отличаются от коннекторов со стороны хаба, чтобы пользователь случайно не подсоединил кабель другой стороной.

Кабель состоит из четырех проводов: два из них предназначены для передачи данных, один — для питания (+5 В) и один — для земли. Система передает 0 изменением напряжения, а 1 — отсутствием изменения напряжения, поэтому длинная последовательность нулевых битов порождает поток регулярных импульсов.

При подключении нового устройства ввода-вывода корневой хаб обнаруживает этот факт и прерывает работу операционной системы. Затем операционная система опрашивает новое устройство, выясняя, что оно собой представляет и какая пропускная способность шины для него требуется. Если операционная система решает, что для этого устройства пропускной способности достаточно, она приписывает ему уникальный адрес (1–127) и загружает этот адрес и другую информацию в конфигурационные регистры внутри устройства. Таким образом, новые устройства могут подсоединяться «на лету», при этом пользователю не нужно устанавливать новые платы ISA или PCI. Неинициализированные платы начинаются с адреса 0, поэтому к ним можно обращаться. Многие устройства снабжены встроенными сетевыми концентраторами для дополнительных устройств. Например, монитор может содержать два хаба для правой и левой колонок.

Шина USB представляет собой ряд каналов между корневым хабом и устройствами ввода-вывода. Каждое устройство может разбить свой канал максимум на 16 подканалов для различных типов данных (например, аудио и видео). В каждом канале или подканале данные перемещаются от корневого хаба к устройству и обратно. Между двумя устройствами ввода-вывода обмена информацией не происходит.

Ровно через каждую миллисекунду (±0,05 мс) корневой хаб передает новый кадр, чтобы синхронизировать все устройства во времени. Кадр состоит из пакетов, первый из которых передается от хаба к устройству. Следующие пакеты кадра могут передаваться в том же направлении, а могут и в противоположном (от устройства к хабу). На рис. 3.54 показаны четыре последовательных кадра.

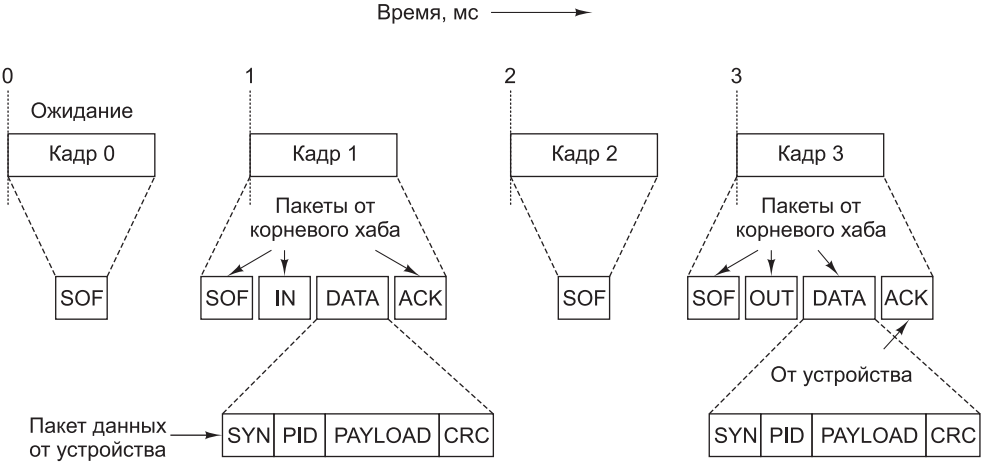


Рис. 3.54. Корневой хаб шины USB передает кадры каждую миллисекунду

В кадрах 0 и 2 не происходит никаких действий, поэтому в них содержится только пакет SOF (Start of Frame — начало кадра). Этот пакет всегда посылается всем устройствам. Кадр 1 — упорядоченный запрос (например, сканеру посылается запрос на передачу битов сканированного им изображения). Кадр 3 состоит из данных, передаваемых какому-нибудь устройству (например, принтеру).

Шина USB поддерживает 4 типа кадров: кадры управления, изохронные кадры, кадры передачи больших массивов данных и кадры прерывания. Кадры управления используются для конфигурирования устройств, передачи устройствам команд и запросов об их состоянии. Изохронные кадры предназначены для устройств реального времени (микрофонов, акустических систем и телефонов), которые должны принимать и посылать данные через равные временны́е интервалы. Задержки хорошо прогнозируются, но в случае ошибки такие устройства не производят повторной передачи. Кадры следующего типа используются для передач большого объема от устройств и к устройствам (например, принтерам) без требований, характерных для устройств реального времени. Наконец, кадры последнего типа нужны потому, что шина USB не поддерживает прерываний. Например, чтобы не вызывать прерывание всякий раз, когда нажимается клавиша, операционная система может опрашивать ее каждые 50 миллисекунд и «собирать» все необработанные данные о нажатии клавиш.

Кадр состоит из одного или нескольких пакетов. Пакеты могут посылаться в обоих направлениях. Существуют четыре типа пакетов: маркеры, пакеты данных, пакеты квитирования и специальные пакеты. Маркеры передаются от концентратора к устройству и предназначены для управления системой. Пакеты SOF, IN и OUT на рис. 3.54 — маркеры. Пакет SOF (Start of Frame — начало кадра) является первым в любом кадре, он идентифицирует начало кадра. Если никаких действий

выполнять не нужно, пакет SOF — единственный в кадре. Пакет IN — это запрос. Этот пакет требует, чтобы устройство выдало определенные данные. Поля в пакете IN содержат информацию о том, какой именно канал запрашивается, и по этой информации устройство определяет, какие именно данные выдавать (если оно манипулирует несколькими потоками данных). Пакет OUT объявляет, что далее последует передача данных для устройства. Последний тип маркера, SETUP (он не показан на рисунке), используется при конфигурировании.

Помимо маркеров существуют еще три типа пакетов. Это пакеты данных (используются для передачи 64 байт информации в обоих направлениях), пакеты квитирования и специальные пакеты. Формат пакета данных (DATA) показан на рис. 3.54. Он состоит из 8-разрядного поля синхронизации, 8-разрядного идентификатора типа пакета (PID), полезной нагрузки и 16-разрядного **CRC-кода** для обнаружения ошибок. Есть три типа пакетов квитирования: ACK (предыдущий пакет данных принят правильно), NAK (найдена ошибка CRC-кода) и STALL (устройство занято, ждите,).

А теперь давайте снова посмотрим на рис. 3.54. Корневой хаб должен отсылать по кадру каждую миллисекунду, даже если не происходит никаких действий. Кадры 0 и 2 содержат только один пакет SOF, который говорит о том, что ничего не происходит. Кадр 1 реализует опрос, поэтому начинается с пакетов SOF и IN, которые передаются от компьютера к устройству ввода-вывода, затем следует пакет DATA от устройства к компьютеру. Пакет ACK сообщает устройству, что данные были получены без ошибок. В случае ошибки устройство получает пакет NACK, после чего данные передаются заново (отметим, что изохронные данные повторно не передаются). Кадр 3 похож по структуре на кадр 1, но в нем поток данных направлен от компьютера к устройству.

После того как в 1998 году стандарт USB был окончательно утвержден, разработчики приступили к созданию следующей, высокоскоростной версии USB, названной **USB 2.0**. Этот стандарт во многом аналогичен USB 1.1 и совместим с ним, однако к двум прежним скоростям в нем добавляется новая — 480 Мбит/с. Все прочие изменения, включая реализацию нового интерфейса между корневым хабом и контроллером, не так существенны. В стандарте USB 1.1 было предусмотрено два интерфейса UHCI и OHCI. Интерфейс **UHCI** (Universal Host Controller Interface — **универсальный интерфейс хост-контроллера**) разработала компания Intel, переложив большую часть забот на программистов (читай — на Microsoft). Программисты вернули должок и выпустили интерфейс **OHCI** (Open Host Controller Interface — **открытый интерфейс хост-контроллера**), взвалив основную работу на разработчиков аппаратуры (читай — Intel). В процессе разработки стандарта USB 2.0 стороны пришли к взаимоприемлемому решению, выпустив новый интерфейс под названием **EHCI** (Enhanced Host Controller Interface — **усовершенствованный интерфейс хост-контроллера**).

Поскольку шина USB теперь передает данные со скоростью 480 Мбит/с, она становится серьезным конкурентом последовательной шины IEEE 1394 (FireWire), работающей на скорости 400 Мбит/с или 800 Мбит/с. Так как почти все современные системы на базе Intel оснащены шиной USB 2.0 или USB 3.0 (см. ниже), стандарт 1394 скоро уйдет в прошлое, причем основной причиной его вымирания будет не устаревание, а борьба за сферы влияния. USB — продукт компьютерной отрасли, а стандарт 1394 появился в мире потребительской электроники. Когда

дело доходит до подключения камеры к компьютеру, каждая сторона желает использовать свой интерфейс. Похоже, «компьютерщики» в этой борьбе победили.

Через восемь лет после выхода USB 2.0 был анонсирован стандарт интерфейса USB 3.0. USB 3.0 поддерживает потрясающую пропускную способность 5 Гбит/с, хотя максимальная скорость, вероятно, будет достигаться только при использовании кабелей профессионального уровня. Устройства USB 3.0 структурно идентичны более ранним устройствам USB, и они полностью реализуют стандарт USB 2.0. Таким образом, при подключении к разъему USB 2.0 устройство USB 3.0 будет работать нормально.

Интерфейсы

Обычная компьютерная система малого или среднего размера состоит из микросхемы процессора, микросхем памяти и нескольких устройств ввода-вывода. Все эти микросхемы соединены шиной. Иногда все эти устройства интегрируются в однокристальную систему, как в случае TI OMAP4430. Мы уже рассмотрели память, центральные процессоры и шины. Теперь настало время изучить интерфейсы ввода-вывода. Именно через эти микросхемы компьютер обменивается информацией с внешними устройствами.

Интерфейсы ввода-вывода

В настоящее время существуют множество различных интерфейсов ввода-вывода, причем постоянно появляются новые интерфейсы. Из наиболее распространенных можно назвать UART, USART, контроллеры CRT, дисковые контроллеры и PIO. **UART** (Universal Asynchronous Receiver Transmitter — **универсальный асинхронный приемопередатчик**) — интерфейс ввода-вывода, который может считать байт из шины данных и побитно передать этот байт в линию последовательной передачи к терминалу или от терминала. Скорость работы микросхем UART различна: от 50 до 19 200 бит/с; ширина символа от 5 до 8 бит; 1, 1,5 или 2 стоповых бита; с проверкой на четность или на нечетность, или без нее — все управляется программно. **USART** (Universal Synchronous Asynchronous Receiver Transmitter — **универсальный синхронно-асинхронный приемопередатчик**) может осуществлять синхронную передачу, используя ряд протоколов, а также поддерживает все функции микросхемы UART. Так как с отмиранием телефонных модемов интерфейс UART уже не играет заметной роли, в качестве примера микросхемы ввода-вывода мы рассмотрим параллельный интерфейс PIO.

Интерфейсы PIO

Типичным примером интерфейса **PIO** (Parallel Input/Output — **параллельный ввод-вывод**) является микросхема Intel 8255A (рис. 3.55). Она содержит 24 линии ввода-вывода и может сопрягаться с любыми устройствами цифровой логики (например, клавиатурами, коммутаторами, индикаторами, принтерами). Программное обеспечение центрального процессора может записать 0 или 1 на любую линию или считать входное состояние любой линии, обеспечивая высокую гибкость. Небольшая система на базе процессора, использующая интерфейс

PIO, может управлять разнообразными физическими устройствами — роботами, тостерами, электронными микроскопами. Чаще всего интерфейсы PIO встречаются во встроенных системах.

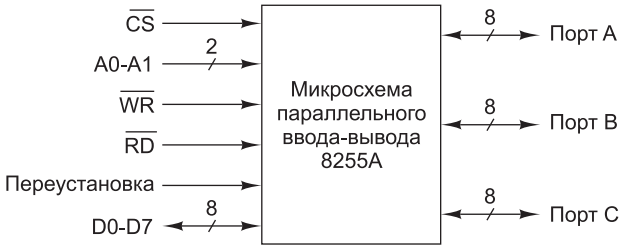


Рис. 3.55. Микросхема 8255A

Конфигурация интерфейса PIO определяется 3-разрядным регистром, который указывает, должны ли три независимых 8-разрядных порта использоваться для ввода (0) или вывода цифрового сигнала (1). Запись соответствующего значения в регистр конфигурации позволяет осуществлять произвольную комбинацию ввода-вывода по трем портам. С каждым портом связан 8-разрядный регистр. Чтобы настроить линии на порт, центральный процессор записывает 8-разрядное число в соответствующий регистр, и это 8-разрядное число появляется на выходных линиях и остается там до тех пор, пока регистр не будет перезаписан. Чтобы использовать порт для ввода, центральный процессор просто считывает соответствующий регистр.

Возможно построение и более сложных интерфейсов PIO. Например, один популярный режим предусматривает квитирование связи с внешними устройствами. Например, чтобы передать данные устройству, микросхема 8255A может передать данные в порт вывода и подождать, пока устройство не выдаст сигнал о том, что данные получены и можно посылать еще. В микросхему включены необходимые логические схемы для фиксации таких сигналов и передачи их центральному процессору.

Из функциональной диаграммы PIO на рис. 3.55 мы видим, что помимо 24 выводов для трех портов микросхема 8255A содержит восемь линий, непосредственно связанных с шиной данных, линию выбора элемента памяти, линии чтения и записи, две адресные линии и линию для переустановки микросхемы. Две адресные линии позволяют выбрать один из четырех внутренних регистров, три из которых соответствуют портам A, B и C, а четвертый регистр является регистром конфигурации. Обычно две адресные линии соединяются с двумя младшими битами адресной шины. Линия выборки позволяет строить из 24-разрядных интерфейсов PIO интерфейсы большей разрядности, с добавлением дополнительных адресных линий и использования их для выбора нужного интерфейса посредством установки линии выборки микросхемы.

Декодирование адреса

До настоящего момента мы не останавливались подробно на том, как происходит выбор микросхемы памяти или устройства ввода-вывода. Рассмотрим простой 16-разрядный встроенный компьютер, состоящий из центрального процессора,

памяти EPROM объемом 2 К × 8 байт для хранения программы, ОЗУ объемом 2 К × 8 байт для хранения данных и интерфейса PIO. Такая небольшая система может встраиваться в дешевую игрушку или простой прибор. После того как продукт пойдет в массовое производство, EEPROM заменяется обычным ПЗУ.

Выборка интерфейса PIO может осуществляться одним из двух способов: как устройства ввода-вывода или как части памяти. Если мы собираемся использовать PIO в качестве устройства ввода-вывода, то мы должны обратиться к нему по соответствующей линии шины, которая означает, что обращение относится к устройству ввода-вывода, а не к памяти. Другой подход называется **вводом-выводом с отображением на память** (memory-mapped I/O). В этом случае микросхеме требуется назначить 4 байта памяти для трех портов и регистра управления. Наш выбор в какой-то степени произволен. Рассмотрим ввод-вывод с отображением на память, поскольку этот подход наглядно иллюстрирует некоторые интересные проблемы сопряжения.

Памяти EPROM требуется 2 Кбайт адресного пространства, ОЗУ — также 2 Кбайт, PIO — 4 байта. Поскольку в нашем примере адресное пространство составляет 64K адресов, мы должны выбрать, где поместить указанные три устройства. Один из возможных вариантов показан на рис. 3.56. EPROM занимает адреса до 2К, ОЗУ — от 32К до 34К, PIO — 4 старших байта адресного пространства, от адресов 65 532 до 65 535. С точки зрения программиста не важно, какие именно адреса использовать, однако для сопряжения это имеет большое значение. Если бы мы обращались к PIO через пространство ввода-вывода, нам не потребовались бы адреса памяти (зато понадобились бы четыре адреса пространства ввода-вывода).

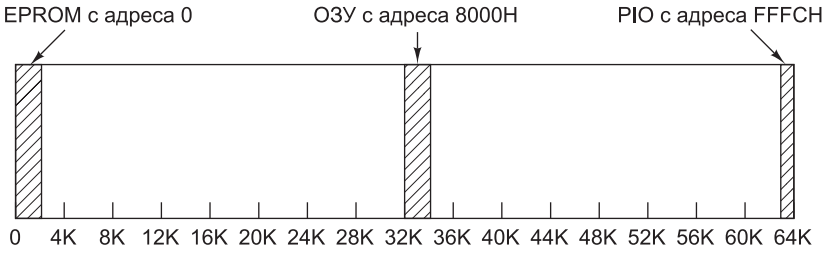


Рис. 3.56. Расположение EPROM, ОЗУ и PIO в пространстве из 64K адресов

При таком распределении адресов к EPROM нужно обращаться с помощью 16-разрядного адреса памяти 00000*xxxxxxxxxxx* (в двоичной системе). Другими словами, любой адрес, у которого пять старших битов равны 0, попадает в область памяти до 2К, то есть в EPROM. Таким образом, сигнал выбора EPROM можно связать с 5-разрядным компаратором, у которого один из входов всегда будет соединен с 00000.

Того же результата можно достичь более эффективно: с использованием вентиля ИЛИ с пятью входами, связанными с адресными линиями от A_{11} до A_{15}. Выходной сигнал может быть равен 0 тогда и только тогда, когда все пять линий равны 0. В этом случае устанавливается сигнал $\overline{CS}$. Этот метод адресации показан на рис. 3.57, *а*; он называется полным декодированием адреса.

Тот же принцип можно применить и для ОЗУ. Однако ОЗУ должно отзываться на бинарные адреса типа 10000*xxxxxxxxxxx*, поэтому необходим дополнительный инвертор (он показан на схеме). Декодирование адреса PIO несколько слож-

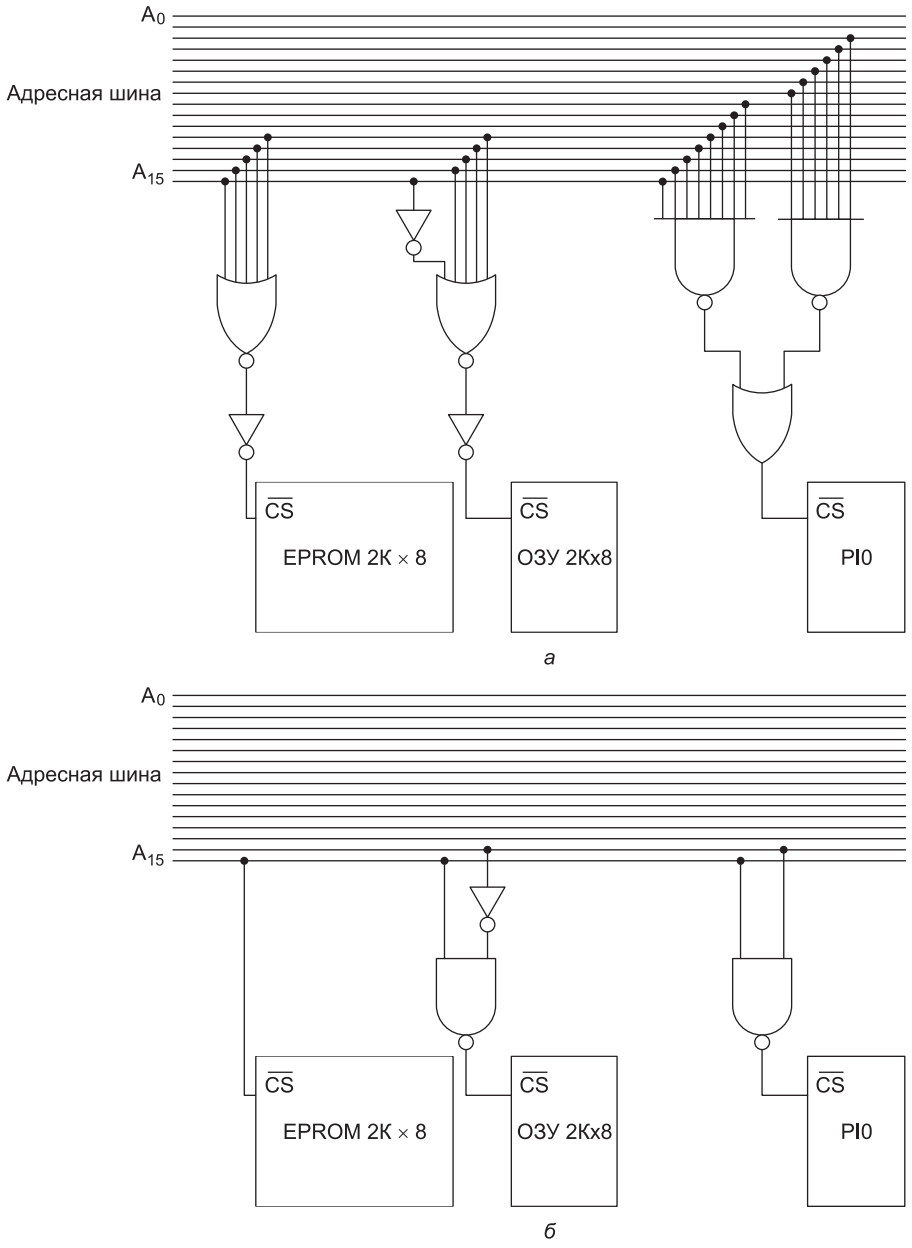


Рис. 3.57. Полное декодирование адреса (*а*); частичное декодирование адреса (*б*)

нее, поскольку он выбирается с помощью 4-х адресов типа 11111111111111*xx*. Один из возможных вариантов схемы, которая устанавливает сигнал $\overline{\text{CS}}$ только в том случае, если на адресной шине появляется адрес данного типа, показан на рисунке. Здесь используются два вентиля НЕ-И с восемью входами, которые соединяются с вентилем ИЛИ.

Если компьютер состоит только из центрального процессора, двух микросхем памяти и PIO, декодирование адреса значительно упрощается. Дело в том, что у всех адресов EPROM (и только у EPROM) старший бит A_{15} всегда равен 0. Следовательно, мы можем просто связать сигнал $\overline{CS}$ с линией A_{15}, как показано на рис. 3.57, *б*.

Теперь решение разместить ОЗУ с адреса 8000H кажется не таким уж произвольным. Отметим, что в ОЗУ попадают адреса типа 10*xxxxxxxxxxxxxx*, поэтому для декодирования достаточно двух бит. Точно так же, любой адрес, начинающийся с 11, является адресом PIO. Теперь полная логика декодирования состоит из двух вентилей НЕ-И и инвертора.

Логика, которую иллюстрирует рис. 3.57, *б*, называется **частичным декодированием адреса**, поскольку в данном случае полные адреса не используются. При таком декодировании считывание из адресов 0001000000000000, 0001100000000000 и 0010000000000000 дает один и тот же результат. В действительности любой адрес в нижней половине адресного пространства означает выбор EPROM. Поскольку дополнительные адреса не используются, в этом нет ничего ужасного, но при разработке компьютера, который в будущем предполагается расширять (в случае с игрушками это маловероятно), следует избегать частичного декодирования, поскольку оно значительно ограничивает адресное пространство.

Можно применять и другую технологию декодирования адреса — с использованием декодера (см. рис. 3.12). Связав три входа с тремя адресными линиями самых старших разрядов, мы получаем восемь выходов, которые соответствуют адресам в первом отрезке 8К, втором отрезке 8К и т. д. В компьютере, содержащем 8 микросхем ОЗУ по 8К × 8 байт, полное декодирование осуществляет одна такая микросхема. Если компьютер содержит 8 микросхем памяти по 2 К × 8 байт, для декодирования также достаточно одного декодера при условии, что каждая микросхема памяти занимает отдельную область адресного пространства в 8К. (Вспомните наше замечание о том, что расположение микросхем памяти и устройств ввода-вывода внутри адресного пространства имеет значение.)

Краткое содержание главы

Компьютеры собираются из интегральных схем, содержащих крошечные переключатели, которые называются вентилями. Обычно используются вентили И, ИЛИ, НЕ-И, НЕ-ИЛИ и НЕ. Комбинируя отдельные вентили, можно строить простые схемы.

Более сложными схемами являются мультиплексоры, демультиплексоры, кодеры, декодеры, схемы сдвига и АЛУ. С помощью программируемой вентильной матрицы (FPGA) можно запрограммировать произвольные булевы функции. Если требуется много булевых функций, программируемые логические матрицы обычно более эффективны, чем другие средства. Для преобразования схем из одной формы в другую используются законы булевой алгебры. Во многих случаях это позволяет создать более экономичные схемы.

Арифметические действия в компьютерах осуществляются сумматорами. Одноразрядный полный сумматор можно сконструировать из двух полусумма-

торов. Чтобы построить сумматор для многоразрядных слов, полные сумматоры соединяются таким образом, чтобы выходной сигнал переноса каждого сумматора передавался его левому соседу.

Статическая память состоит из защелок и триггеров, каждый из которых может хранить один бит информации. Их можно объединять, получая восьмиразрядные триггеры и защелки или готовую память для хранения слов. Существуют различные типы памяти: ОЗУ, ПЗУ, PROM, EPROM, EEPROM, флэш-память. Статическое ОЗУ не нужно обновлять: оно хранит информацию, пока включен компьютер. Динамическое ОЗУ, напротив, нужно периодически обновлять, чтобы предотвратить потерю информации.

Компоненты компьютерной системы соединяются шинами. Большинство выводов обычного центрального процессора (хотя не все) образуют одну линию шины. Линии шины можно подразделить на адресные, информационные и управляющие. Синхронные шины управляются задающим генератором. В асинхронных шинах для согласования работы задающего и подчиненного устройств используется система полного квитирования.

Core i7 представляет собой пример современного процессора. Системы с таким процессором включают в себя шину памяти, шину PCI и шину USB. Шина PCIe — популярный механизм связывания внутренних компонентов компьютера, работающих на высоких скоростях. ARM также входит в категорию современных высокопроизводительных процессоров, но основной областью применения ARM являются встроенные и мобильные устройства, для которых важно низкое потребление энергии. Atmel ATmega168 — пример недорогой микросхемы для компактных, бюджетных устройств и других приложений, критичных по цене.

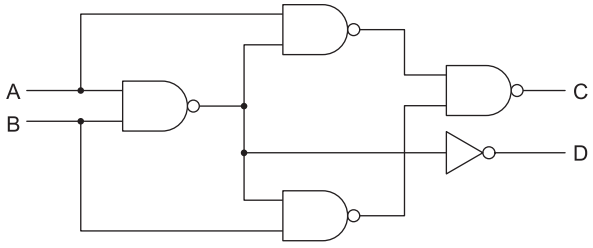
Коммутаторы, индикаторы, принтеры и многие другие устройства ввода-вывода можно связать с компьютером, используя параллельные интерфейсы ввода-вывода. Эти микросхемы по желанию можно сделать частью пространства ввода-вывода или частью пространства памяти. При обращениях к микросхеме может использоваться полное или частичное декодирование адреса в зависимости от того, какие задачи решает компьютер.

Вопросы и задания

1. Аналоговые схемы подвержены воздействию шума, искажающего их вывод. Защищены ли от шума цифровые устройства? Аргументируйте ответ.
2. Логик заезжает в закусочную и говорит: «Дайте мне, пожалуйста, гамбургер или хот-дог и картофель фри». К несчастью, повар не закончил и шести классов и не знает (да и не хочет знать), какая из двух логических операций, И или ИЛИ, имеет приоритет над другой. Он считает, что в данном случае допустима любая интерпретация. А какие из перечисленных здесь вариантов интерпретации этого высказывания действительно допустимы? (Обратите внимание, что ИЛИ здесь трактуется как ИСКЛЮЧАЮЩЕЕ ИЛИ.)
   1) Только гамбургер.
   2) Только хот-дог.
   3) Только картофель фри.

4) Хот-дог и картофель фри.
5) Гамбургер и картофель фри.
6) Хот-дог и гамбургер.
7) Все три.
8) Ничего — логик останется голодным, потому что он слишком умный.

3. Миссионер, заблудившийся в Южной Калифорнии, остановился на развилке дороги. Он знает, что в этом районе обитают две мотоциклетные банды. Одна из них всегда говорит правду, а другая всегда лжет. Он хочет узнать, какая дорога ведет в Диснейленд. Какой вопрос он должен задать?
4. Используя таблицу истинности покажите, что $X = (X \text{ И } Y) \text{ ИЛИ } (X \text{ И НЕ } Y)$.
5. Существует 4 булевы функции от одной переменной и 16 функций от двух переменных. Сколько существует функций от трех переменных? А от n переменных?
6. Существует 4 булевы функции от одной переменной и 16 функций от двух переменных. Сколько существует функций от четырех переменных?
7. Покажите, как можно реализовать функцию И, используя два вентиля НЕ-И.
8. Используя мультиплексор с тремя переменными, изображенный на рис. 3.11, реализуйте функцию, значение которой равно 1 тогда и только тогда, когда нечетное число входных сигналов равно 1.
9. Мультиплексор с тремя переменными, изображенный на рис. 3.11, в действительности способен вычислять произвольную функцию от *четырех* логических переменных. Опишите, как это происходит, и нарисуйте логическую схему для функции, которая принимает значение 0, если слово, соответствующее строке таблицы истинности, содержит четное число букв, и 1, если оно содержит нечетное число букв (например, 0000 = нуль = четыре буквы → 0; 0010 = два = три буквы → 1; 0111 = семь = четыре буквы → 0; 1101 = тринадцать = десять букв → 0). *Подсказка*: назовем четвертую входную переменную D. Тогда восемь входных линий можно связать с V_{CC}, землей, D или $\overline{D}$.
10. Нарисуйте логическую схему двухразрядного кодера, который содержит 4 входные и 2 выходные линии. Одна из входных линий всегда равна 1. Двухразрядное двоичное число на двух выходных линиях показывает, какая именно входная линия равна 1.
11. Нарисуйте логическую схему двухразрядного демультиплексора, у которого сигнал на единственной входной линии направляется к одной из четырех выходных линий в зависимости от значений двух линий управления.
12. Что делает данная схема?



13. 4-разрядный сумматор — вполне распространенная микросхема. Четыре такие схемы можно связать вместе и получить 16-разрядный сумматор. Как вы думаете, сколько выводов должен содержать каждый 4-разрядный сумматор? Почему?
14. Сумматор с числом разрядов n можно получить путем каскадного объединения n полных сумматоров, причем перенос в стадию i, который мы обозначим C_i, получается из результата вычислений на стадии $i - 1$. Перенос в стадию 0, C_0 равен 0. Если вычисление суммы и переноса составляет на каждой стадии T нс, то перенос в стадию i будет вычислен только через iT нс после начала суммирования. При большом n до вычисления переноса в последнюю стадию может пройти очень много времени. Разработайте сумматор, который работает быстрее. *Подсказка*: каждый перенос C_i можно выразить через операнды (биты) A_{i-1} и B_{i-1}, так же как и перенос C_{i-1}. Используя это соотношение, можно выразить C_i как функцию от входных сигналов на стадии от 0 до $i - 1$, так что все переносы можно будет генерировать одновременно.
15. Если все вентили на рис. 3.17 имеют задержку на прохождение сигнала в 1 нс, а все прочие задержки не учитываются, сколько потребуется времени (минимум) для получения достоверного выходного сигнала?
16. АЛУ, изображенное на рис. 3.18, способно выполнять сложение 8-разрядных двоичных чисел. Может ли оно выполнять вычитание двоичных чисел? Если да, то объясните, как. Если нет, преобразуйте схему таким образом, чтобы она могла вычитать.
17. 16-разрядное АЛУ собирается из 16 одноразрядных АЛУ, каждое из которых тратит на суммирование 10 нс. Если задержка на прохождение сигнала от одного АЛУ к другому составляет 1 нс, сколько времени потребуется для получения конечного результата?
18. Иногда требуется, чтобы 8-разрядное АЛУ (см., например, рис. 3.18) выдавало на выходе константу –1. Предложите два различных способа того, как это можно сделать. Для каждого способа определите значения шести сигналов управления.
19. Каково состояние покоя входов S и R у SR-защелки, построенной из двух вентилей НЕ-И?
20. Схема на рис. 3.24 представляет собой триггер, который запускается на фронте синхронизирующего сигнала. Преобразуйте эту схему так, чтобы получить триггер, который запускается на спаде синхронизирующего сигнала.
21. В памяти размером 4 × 3, изображенной на рис. 3.27, используются 22 вентиля И и три вентиля ИЛИ. Сколько потребуется вентилей каждого из двух типов, если схему расширить до размера 256 × 8?
22. Вы консультируете неопытных производителей микросхем. Один из ваших клиентов по требованию потенциального важного покупателя предложил выпустить микросхему, содержащую четыре D-триггера, каждый из которых имеет выходы Q и $\overline{Q}$. В данном проекте все 4 синхронизирующих сигнала объединены (также по требованию покупателя). Входов предварительной установки и очистки у схемы нет. Ваша задача — дать профессиональную оценку этой разработке.

23. С увеличением объема памяти, помещаемой на одну микросхему, число выводов, необходимых для обращения к этой памяти, также увеличивается. Иметь большое количество адресных выводов на микросхеме довольно неудобно. Придумайте способ обращения к 2^n словам памяти при наличии меньшего количества выводов, чем n.
24. В компьютере с 32-разрядной шиной данных используются динамические ОЗУ размером 1 М × 1. Каков минимальный объем памяти (в байтах), который может содержаться в этом компьютере?
25. Вернитесь к временно́й диаграмме на рис. 3.35. Предположим, вы замедлили задающий генератор до 20 нс вместо 10 нс, но временные ограничения сохранились без изменений. Сколько времени в худшем случае будет у памяти на то, чтобы передать данные в шину во время T_3 после того, как установлен сигнал $\overline{\text{MREQ}}$?
26. Снова посмотрите на рис. 3.35. Предположим, тактовый генератор работает с частотой 100 МГц, а время T_{DS} возросло до 4 нс. Можно ли при этом продолжать использовать микросхемы памяти на 10 нс?
27. В таблице на рис. 3.35, *б* показано, что значение T_{ML} должно быть по крайней мере 2 нс. Можете ли вы представить микросхему, у которой этот показатель отрицательный? Другими словами, может ли процессор устанавливать сигнал $\overline{\text{MREQ}}$ до выяснения адреса? Объясните, почему.
28. Предположим, что передача блока на рис. 3.39 была произведена на шине, временна́я диаграмма которой показана на рис. 3.35. Насколько больше получается пропускная способность при передаче блока по сравнению с отдельными передачами (для длинных блоков)? А теперь предположите, что ширина шины составляет не 8, а 32 бита. Каков будет ваш ответ теперь?
29. Посмотрите на рис. 3.36. Обозначьте время передачи адресных линий как T_{A1} и T_{A2}, время передачи линии $\overline{\text{MREQ}}$ как T_{MREQ1} и T_{MREQ2} и т. д. Напишите все неравенства, подразумеваемые при полном квитировании.
30. Сейчас становятся популярными многоядерные микросхемы с несколькими процессорами на одной подложке. Какими преимуществами обладает такая архитектура по сравнению с системой, состоящей из нескольких PC, соединенных по сети Ethernet?
31. Почему появились многоядерные процессоры? Какие технологические факторы проложили им путь? Играет ли закон Мура какую-либо роль в их появлении?
32. Чем шина памяти отличается от шины PCI?
33. Большинство 32-разрядных шин допускают считывание и запись порциями по 16 бит. Существует ли неоднозначность относительно того, куда следует поместить данные? Аргументируйте.
34. Многие процессоры поддерживают особый тип цикла шины для подтверждения прерывания. Зачем это нужно?
35. 32-разрядному компьютеру с шиной, работающей на частоте 200 МГц, требуется 4 цикла, чтобы считать 32-разрядное слово. Какую часть пропускной способности шины потребляет процессор в наихудшем случае?

36. 64-разрядному компьютеру с шиной, работающей на частоте 200 МГц, требуется 4 цикла, чтобы считать 64-разрядное слово. Какую часть пропускной способности шины потребляет процессор в наихудшем случае?
37. 32-разрядный процессор с адресными линиями A_2–A_{31} требует, чтобы все ссылки к ячейкам памяти были выровнены. Это значит, что центральный процессор должен обращаться только к словам, состоящим из 4, 8, 12 и т. д. байт (число байтов кратно 4), и к полусловам, состоящим из четного числа байтов. Байты могут располагаться где угодно. Сколько существует допустимых комбинаций операций считывания из памяти и сколько требуется выводов, чтобы их реализовать? Дайте два ответа и приведите доводы в пользу каждого.
38. Современные процессоры используют один, два и даже три уровня внутреннего кэша. Почему нужна многоуровневая организация кэша?
39. Предположим, что центральный процессор содержит кэш-память первого и второго уровней с временем доступа 1 нс и 2 нс соответственно. Время доступа к основной памяти составляет 10 нс. Если 20 % всех обращений к памяти приходится на долю кэш-памяти первого уровня, а 60 % — на долю кэш-памяти второго уровня, то каково среднее время доступа?
40. Вычислите пропускную способность шины, необходимую для воспроизведения цветного фильма (1280 × 960) с частотой 30 кадров/с. Предполагается, что данные должны проходить по шине дважды: один раз — от компакт-диска к памяти, второй — от памяти к монитору.
41. Какие из сигналов, показанных на рис. 3.51, не являются строго обязательными для протокола шины?
42. Суммарная пропускная способность всех каналов в PCI Express составляет 10 Мбит/с. Сколько сигнальных проводов в каждом направлении потребуется для работы на скорости 16x? Какова будет суммарная пропускная способность в каждом направлении? Полезная пропускная способность в каждом направлении?
43. Компьютеру на выполнение каждой команды требуется два цикла шины: один — для выборки команды, второй — для выборки данных. Каждый цикл шины занимает 10 нс, а выполнение каждой команды занимает 20 нс (время обработки не принимается в расчет). В компьютере имеется диск. Каждая дорожка этого диска состоит из 2048 секторов по 512 байт. Время обращения диска составляет 5 мс. На сколько процентов снижается скорость работы компьютера в случае прямого доступа к памяти, если каждая 32-разрядная операция такого доступа занимает один цикл шины?
44. Максимальная полезная нагрузка пакета данных, передаваемого по шине USB, составляет 1023 байта. Если предположить, что устройство может посылать только один пакет данных за кадр, какова максимальная пропускная способность для одного изохронного устройства?
45. Посмотрите на рис. 3.57, *б*. Что получится, если к вентилю НЕ-И, который позволяет выбрать микросхему PIO, добавить третью входную линию, связанную с A_{13}?

46. Напишите программу, которая имитирует работу матрицы размером $m \times n$, состоящей из двухвходовых вентилей НЕ-И. Эта схема (она помещается на микросхему) содержит j входных и k выходных выводов. Значения j, k, m и n определяются параметрами компиляции модели. Программа считывает таблицу монтажных соединений, каждое из соединений определяет вход и выход. Входом может быть либо один из j входных выводов, либо выход какого-нибудь вентиля НЕ-И. Выходом может быть либо один из k выходных выводов, либо вход в какой-нибудь вентиль НЕ-И. Неиспользованные входы принимают значение логической единицы. После считывания таблицы соединений программа должна напечатать выходное значение для каждого из 2^j возможных входных значений. Подобные вентильные матрицы широко используются при рисовании схем по техническим заданиям заказчика, поскольку большая часть этой работы (имеется в виду нанесение вентильной матрицы на микросхему) не зависит от того, какая это будет схема. Для каждой разработки имеет значение только выбор монтажных соединений.
47. Напишите программу, которая на входе получает два произвольных логических выражения и проверяет, представляют ли они одну и ту же функцию. Входной язык должен включать отдельные буквы (логические переменные), операнды И, ИЛИ и НЕ и скобки. Каждое выражение должно помещаться на одну входную линию. Программа вычисляет таблицы истинности для обеих функций и сравнивает их.

Глава 4
Уровень микроархитектуры

Над цифровым логическим уровнем находится уровень микроархитектуры. Его задача — интерпретация команд уровня 2 (уровня архитектуры команд), как показано на рис. 1.2. Строение уровня микроархитектуры зависит от того, каков уровень архитектуры команд, а также от стоимости и назначения компьютера. В настоящее время на уровне архитектуры команд обычно находятся простые команды, которые выполняются за один цикл (таковы, в частности, RISC-машины). В других системах (например, в Core i7) на этом уровне имеются более сложные команды; выполнение одной такой команды занимает несколько циклов. Чтобы выполнить команду, нужно найти операнды в памяти, считать их и записать полученные результаты обратно в память. Управление уровнем команд со сложными командами отличается от управления уровнем команд с простыми командами, так как в первом случае выполнение одной команды требует определенной последовательности операций.

Пример микроархитектуры

В идеале неплохо было бы сначала описать общие принципы разработки уровня микроархитектуры, но, к сожалению, таких общих принципов не существует. Каждая разработка индивидуальна. По этой причине мы просто подробно рассмотрим конкретный пример. В качестве примера мы выбрали подмножество виртуальной машины Java. Это подмножество содержит только целочисленные команды, поэтому мы назвали его **IJVM** (Integer Java Virtual Machine — виртуальная машина Java для целых).

Начнем мы с описания микроархитектуры, на базе которой воплотим IJVM. IJVM содержит несколько довольно сложных команд. Как уже отмечалось в главе 1, подобные архитектуры часто реализуются путем микропрограммирования. Хотя структура IJVM не слишком сложная, она может стать хорошей отправной точкой при описании основных принципов обработки команд и последовательностью их выполнения.

Наша микроархитектура содержит микропрограмму (в ПЗУ), которая должна вызывать, декодировать и выполнять IJVM-команды. Мы не можем использовать для этой микропрограммы интерпретатор Oracle JVM, поскольку нам нужна всего лишь крошечная микропрограмма, которая запускает отдельные вентили аппаратного обеспечения. Интерпретатор Oracle JVM был написан на языке C для обеспечения портируемости программного обеспечения, и он не способен управлять аппаратным обеспечением.

Поскольку реальное аппаратное обеспечение состоит только из компонентов, описанных в главе 3, то теоретически после прочтения этой главы читатель сможет пойти в магазин, купить мешок транзисторов и сконструировать машину

IJVM. Тому, кто успешно выполнит это задание, полагаются дополнительные баллы (а также обследование у психиатра).

Условимся каждую команду уровня архитектуры команд считать функцией, вызываемой из основной программы. В данном случае основная программа довольно проста. Она представляет собой бесконечный цикл. Сначала программа определяет, какую функцию нужно выполнить, затем вызывает эту функцию, после чего все снова повторяется — почти как на рис. 2.3.

Микропрограмма содержит набор переменных, к которым имеют доступ все функции. Этот набор переменных называется **состоянием** компьютера. Каждая функция изменяет по крайней мере несколько переменных, формируя при этом новое состояние. Например, счетчик команд — это часть состояния. Он указывает местонахождение очередной функции (то есть команды уровня архитектуры команд), которая должна быть выполнена. Во время выполнения каждой команды счетчик команд указывает на следующую команду.

IJVM-команды очень короткие. Каждая команда состоит из нескольких полей, обычно одного или двух, каждое из которых решает определенную задачу. Первое поле содержит **код операции**. Этот код задает тип команды (например, сложение, переход или еще какая-нибудь команда). Многие команды содержат дополнительное поле, которое определяет тип операнда. Например, команды, которые имеют доступ к локальным переменным, должны иметь специальное поле, чтобы определить, *какая* это переменная.

Такая модель выполнения команды, называемая иногда **циклом выборка-декодирование-исполнение**, полезна для теории и может стать основой воплощения уровня архитектуры команд со сложными командами (например, IJVM). Далее мы опишем, как работает эта модель, что собой представляет микроархитектура и как ею управляют микрокоманды, каждая из которых занимает тракт данных на один цикл. Полный список команд формирует микропрограмму, которая будет рассмотрена очень подробно.

Тракт данных

Тракт данных — это часть центрального процессора, состоящая из АЛУ (арифметико-логического устройства), его входов и выходов. Тракт данных нашей микроархитектуры показан на рис. 4.1. Хотя этот тракт данных и был оптимизирован для интерпретации IJVM-программ, он схож с трактами данных большинства компьютеров. Тракт содержит ряд 32-разрядных регистров, которым мы приписали символические названия (например, PC, SP, MDR). Хотя некоторые из этих названий нам знакомы, важно понимать, что эти регистры доступны только на уровне микроархитектуры (для микропрограммы). Им даны такие названия, поскольку они обычно содержат значения, соответствующие переменным с аналогичными названиями на уровне архитектуры команд. Содержание большинства регистров передается на шину B. Выходной сигнал АЛУ управляет схемой сдвига и далее шиной C. Значение с шины C может записываться в один или несколько регистров одновременно. Шину A мы введем позже, а пока представим, что ее нет.

Данное АЛУ идентично тому, которое изображено на рис. 3.17 и рис. 3.18. Его функционирование зависит от линий управления. На рис. 4.1 перечеркнутая стрелочка с цифрой 6 сверху указывает на наличие шести линий управления

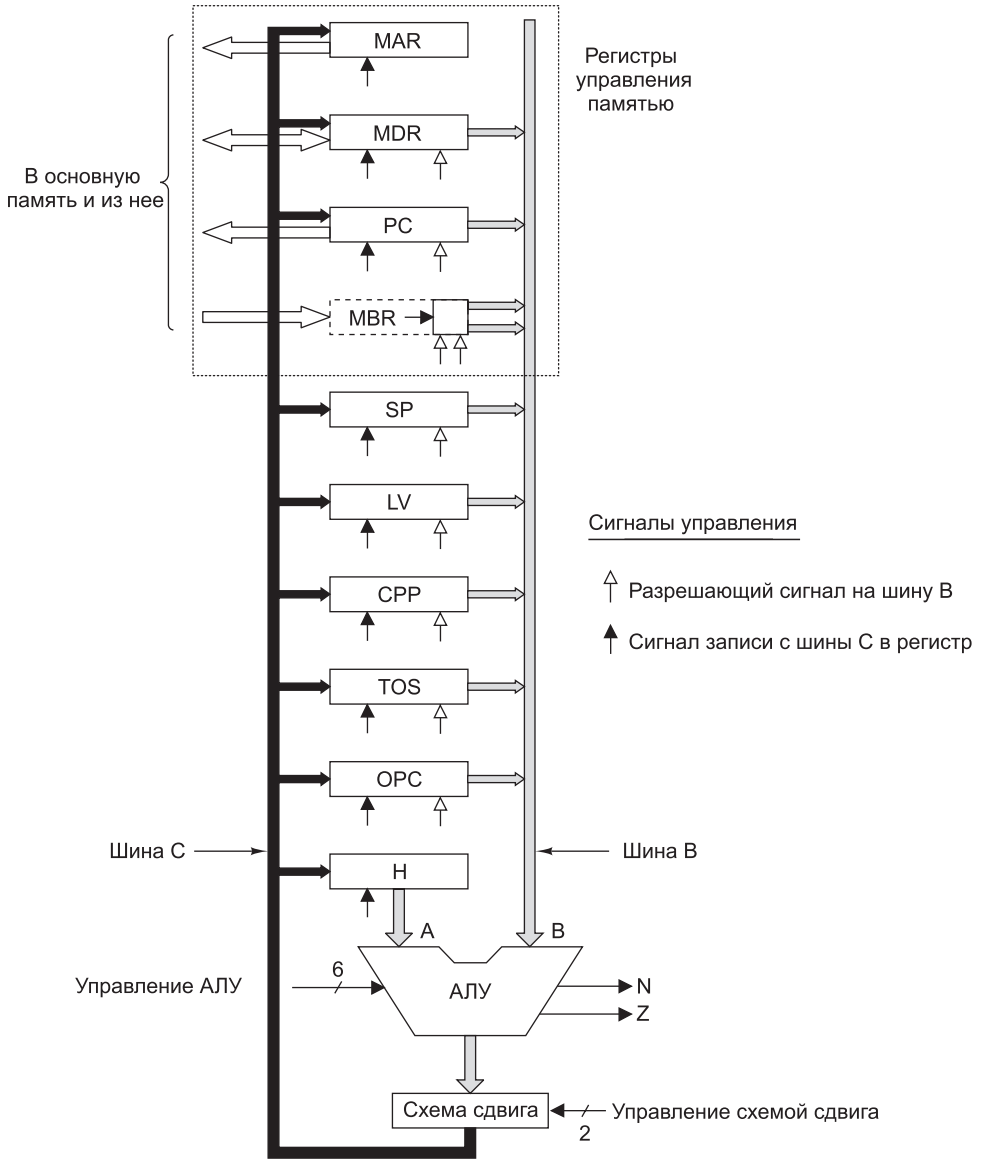


Рис. 4.1. Тракт данных для микроархитектуры, рассматриваемой в этой главе

АЛУ. Из них F_0 и F_1 служат для задания операции, ENA и ENB — для разрешения входных сигналов A и B соответственно, INVA — для инверсии левого входа и INC — для переноса бита в младший разряд, что эквивалентно прибавлению единицы к результату. Впрочем, не все 64 комбинации значений на линиях управления делают что-то полезное.

Некоторые комбинации показаны в табл. 4.1. Не все из этих функций нужны для машины IJVM, но многие из них могут пригодиться для полнофункциональной машины (JVM). В большинстве случаев существуют несколько возможно-

стей для достижения одного и того же результата. В данной таблице знак плюс (+) означает арифметический плюс, а знак минус (–) — арифметический минус, поэтому –A означает дополнение A.

Таблица 4.1. Некоторые комбинации сигналов АЛУ и соответствующие им функции

F_0	F_1	ENA	ENB	INVA	INC	Функция
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	0	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B — A
1	1	0	1	1	0	B — 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A И B
0	1	1	1	0	0	A ИЛИ B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

АЛУ, изображенное на рис. 4.1, содержит два входа для данных: левый вход (A) и правый вход (B). С левым входом связан регистр временного хранения H. С правым входом связана шина B, на которую могут поступать значения с одного из девяти источников, что показано с помощью девяти серых стрелок, примыкающих к шине. Существует и другая разработка АЛУ с двумя полноразрядными шинами, мы рассмотрим ее чуть позже в этой главе.

В регистр H может поступать функция АЛУ, которая проходит через правый вход (из шины B) к выходу АЛУ. Одна из таких функций — сложение входных сигналов АЛУ, только при этом сигнал ENA отрицателен, и левый вход получает значение 0. Если к значению шины B прибавить 0, это значение не изменится. Затем результат проходит через схему сдвига (также без изменений) и сохраняется в регистре H.

Существует еще две линии управления, которые используются независимо от остальных. Они служат для управления выходом АЛУ. Линия SLL8 (Shift Left Logical — логический сдвиг влево) сдвигает число влево на 1 байт, заполняя 8 самых младших двоичных разрядов нулями. Линия SRA1 (Shift Right Arithmetic — арифметический сдвиг вправо) сдвигает число вправо на 1 бит, оставляя самый старший двоичный разряд без изменений.

Операции чтения и записи регистра могут выполняться за один цикл. Для этого, например, нужно поместить значение SP на шину B, закрыть левый вход АЛУ, установить сигнал INC и сохранить полученный результат в регистре SP, увеличив таким образом его значение на 1 (восьмая строка табл. 4.1). Если один и тот же регистр может считываться и записываться за один цикл, то как при этом предотвратить искажение данных? Дело в том, что процессы чтения и записи происходят в разных частях цикла. Когда в качестве правого входа АЛУ выбирается один из регистров, его значение помещается на шину B в начале цикла и хранится там на протяжении всего цикла. Затем АЛУ выполняет свою работу, результат которой через схему сдвига поступает на шину C. Незадолго до конца цикла, когда значения выходных сигналов АЛУ и схемы сдвига стабилизируются, содержание шины C передается в один или несколько регистров. Одним из этих регистров вполне может быть тот, с которого поступил сигнал на шину B. Точная синхронизация тракта данных делает возможным считывание и запись одного и того же регистра за один цикл. Об этом речь пойдет далее.

Синхронизация тракта данных

На рис. 4.2 показано, как происходит синхронизация этих действий. В начале каждого цикла генерируется короткий импульс. Он может выдаваться задающим генератором, как показано на рис. 3.19, *в*. На спаде импульса устанавливаются биты, которые будут запускать все вентили. Этот процесс занимает определенный отрезок времени Δw. Затем выбирается регистр, и его значение передается на шину B. На это требуется время Δx. Далее АЛУ и схема сдвига начинают оперировать поступившими к ним данными. После промежутка Δy выходные сигналы АЛУ и схемы сдвига стабилизируются. В течение следующего отрезка Δz результаты проходят по шине C к регистрам, куда они загружаются на фронте следующего импульса. Загрузка должна запускаться фронтом сигнала и

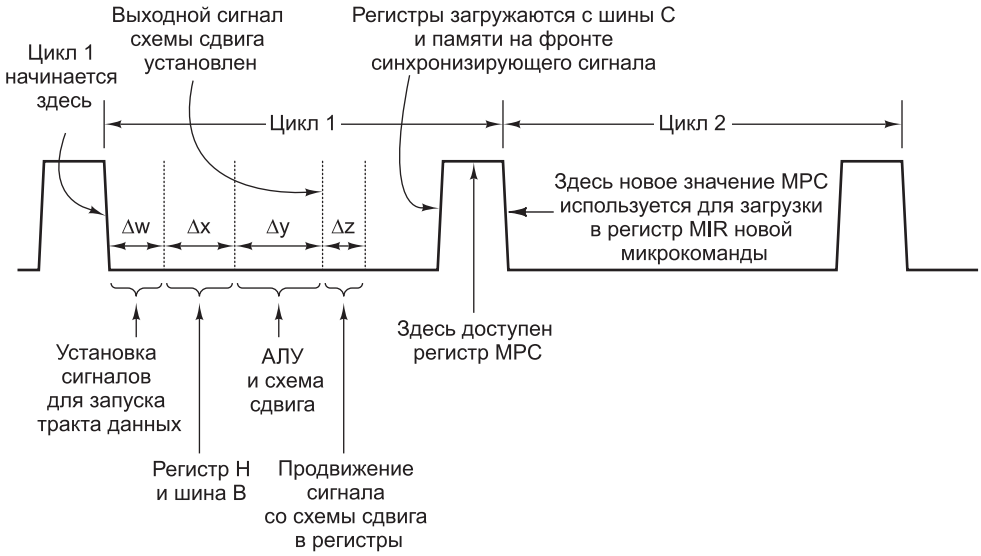


Рис. 4.2. Временна́я диаграмма цикла тракта данных

осуществляться мгновенно, так что даже в случае изменений каких-либо входных регистров изменения в шине C будут происходить только после полной загрузки регистров. На фронте импульса регистр, запускающий шину B, приостанавливает свою работу и ждет следующего цикла. На рисунке имеются еще регистры MPC и MIR, а также память. Их предназначение мы обсудим чуть позже.

Важно понимать, что хотя в тракте данных нет никаких запоминающих элементов, для прохождения сигнала по нему требуется определенное время. Изменение значения на шине B вызывает изменения на шине C не сразу, а только после паузы (это объясняется задержками на каждом шаге). Следовательно, даже если один из входных регистров изменяется, новое значение сохраняется в регистре задолго до того, как старое (и уже неправильное) значение этого регистра, помещенное на шину B, сможет достичь АЛУ.

Чтобы такая система была работоспособна, требуется жесткая синхронизация и довольно длинный цикл; должно быть известно минимальное время прохождения сигнала через АЛУ; регистры должны загружаться с шины C очень быстро. Если подойти к этому вопросу с достаточным вниманием, можно сделать так, чтобы тракт данных функционировал правильно.

Также цикл тракта данных можно рассматривать как совокупность подциклов. Начало подцикла 1 инициируется спадом синхронизирующего сигнала. Далее показано, что происходит во время каждого из подциклов. В скобках приводится величина подцикла.

1. Устанавливаются сигналы управления (Δw).
2. Значения регистров загружаются на шину B (Δx).
3. Действуют АЛУ и схемы сдвига (Δy).
4. Результаты проходят по шине C обратно к регистрам (Δz).

На фронте следующего цикла результаты сохраняются в регистрах.

Никаких внешних сигналов, указывающих на начало и конец подцикла и сообщающих АЛУ, когда нужно начинать работу и передавать результаты на шину C, нет. В действительности АЛУ и схема сдвига работают постоянно. Однако их входные сигналы остаются недействительными до истечения периода $\Delta w + \Delta x$ после спада синхронизирующего сигнала. Точно так же их выходные сигналы недействительны в течение периода $\Delta w + \Delta x + \Delta y$. Единственными внешними сигналами, управляющими трактом данных, являются спад синхронизирующего сигнала, с которого начинается цикл тракта данных, и фронт синхронизирующего сигнала, инициирующий загрузку регистров с шины C. Границы подциклов определяются только временем прохождения сигнала, поэтому для надежной загрузки регистра разработчики тракта данных должны очень четко рассчитать, чтобы время $\Delta w + \Delta x + \Delta y + \Delta z$ стабильно предшествовало фронту синхронизирующего сигнала.

Функционирование памяти

Наша машина может взаимодействовать с памятью двумя способами: через порт с пословной адресацией (32-разрядный) и через порт с побайтовой адресацией (8-разрядный). Порт с пословной адресацией управляется двумя регистрами: **MAR** (Memory Address Register — **адресный регистр памяти**) и **MDR** (Memory Data Register — информационный регистр памяти), которые показаны

на рис. 4.1. 8-разрядный порт управляется регистром PC, который записывает 1 байт в 8 младших битов регистра **MBR** (Memory Buffer Register — **буферный регистр памяти**). Этот порт может считывать данные из памяти, но не может записывать их в память.

Каждый из этих регистров, а также все остальные регистры, изображенные на рис. 4.1, запускаются одним из **сигналов управления**. Белая стрелка под регистром указывает на сигнал управления, который разрешает передавать выходной сигнал регистра на шину B. Регистр MAR не связан с шиной B, поэтому у него нет разрешающего сигнала управления. У регистра H этого сигнала тоже нет, так как он является единственным возможным левым входом АЛУ и поэтому всегда разрешен.

Черная стрелка под регистром указывает на сигнал управления, который записывает (то есть загружает) регистр с шины C. Поскольку регистр MBR не может загружаться с шины C, у него нет записывающего сигнала управления (но зато есть два сигнала разрешения, о которых речь пойдет далее). Чтобы инициировать процесс считывания из памяти или записи в память, нужно загрузить соответствующие регистры памяти, а затем передать памяти сигнал чтения или записи (он не показан на рис. 4.1).

Регистр MAR содержит адреса *слов*, таким образом, значения 0, 1, 2 и т. д. указывают на последовательные слова. Регистр PC содержит адреса *байтов*, таким образом, значения 0, 1, 2 и т. д. указывают на последовательные байты. Если значение 2 поместить в регистр PC и начать процесс чтения, то из памяти считается байт 2, который затем будет записан в 8 младших битов регистра MBR. Если значение 2 поместить в регистр MAR и начать процесс чтения, то из памяти считаются байты с 8 по 11 (то есть слово 2), которые затем будут записаны в регистр MDR.

Для чего потребовалось два регистра с разной адресацией? Дело в том, что регистры MAR и PC будут использоваться для обращения к двум разным областям памяти. Зачем это нужно, станет ясно чуть позже. А пока достаточно сказать, что регистры MAR и MDR используются для чтения и записи слов данных на уровне архитектуры команд, а регистры PC и MBR — для считывания программы уровня архитектуры команд, которая состоит из потока байтов. Во всех остальных регистрах, содержащих адреса, применяется принцип пословной адресации, как и в MAR.

В физической реализации существует только одна память с байтовой адресацией. Как же регистр MAR обращается к словам, если память состоит из байтов? Когда значение регистра MAR помещается на адресную шину, 32 бита этого значения не попадают точно на 32 адресные линии (с 0 по 31). Вместо этого бит 0 соединяется с адресной линией 2, бит один — с адресной линией 3 и т. д. Два старших бита не учитываются, поскольку они нужны только для адресов свыше 2^{32}, а такие адреса недопустимы в нашей машине на 4 Гбайт. Когда значение MAR равно 1, на шину помещается адрес 4; когда значение MAR равно 2, на шину помещается адрес 8 и т. д. Распределение битов регистра MAR по адресным линиям иллюстрирует рис. 4.3.

Как уже отмечалось, данные, считанные из памяти через 8-разрядный порт, сохраняются в 8-разрядном регистре MBR. Этот регистр может быть скопирован на шину B двумя способами: со знаком и без знака. Когда требуется значение

Рис. 4.3. Распределение битов регистра MAR в адресной шине

без знака, 32-разрядное слово, помещаемое на шину B, содержит значение MBR в младших 8-ми битах и нули в остальных 24-х битах. Значения без знака нужны для индексирования таблиц или получения целого 16-разрядного числа из двух последовательных байтов (без знака) в потоке команд.

Другой способ превращения 8-разрядного регистра MBR в 32-разрядное слово — считать его значением со знаком от –128 до +127 включительно и использовать это значение для порождения 32-разрядного слова с тем же самым численным значением. Это преобразование делается путем дублирования знакового (самого левого) бита регистра MBR в верхние 24 битовые позиции шины B. Такой процесс называется **расширением по знаку**, или **знаковым расширением**. Если выбран данный параметр, то либо все старшие 24 бита примут значение 0, либо все они примут значение 1 в зависимости от того, каков самый левый бит регистра MBR: 0 или 1.

В какое именно 32-разрядное значение (со знаком или без знака) превратится 8-разрядное значение регистра MBR, определяется тем, какой из двух сигналов управления (две белые стрелки под регистром MBR на рис. 4.1) установлен. Пунктирный прямоугольник обозначает способность 8-разрядного регистра MBR действовать в качестве источника 32-разрядных слов для шины B.

Микрокоманды

Для управления трактом данных, изображенном на рис. 4.1, необходимо 29 сигналов. Их можно разделить на пять функциональных групп:

- 9 сигналов для записи данных с шины C в регистры;
- 9 сигналов для разрешения передачи регистров на шину B и в АЛУ;
- 8 сигналов для управления АЛУ и схемой сдвига;
- 2 сигнала, которые указывают, что нужно осуществить чтение или запись через регистры MAR/MDR (на рисунке они не показаны);
- 1 сигнал, который указывает, что нужно осуществить вызов из памяти через регистры PC/MBR (на рисунке также не показан).

Значения этих 29 сигналов управления определяют операции для одного цикла тракта данных. Цикл состоит из передачи значений регистров на шину B, прохождения этих сигналов через АЛУ и схему сдвига, передачи полученных

результатов на шину C и записи их в нужный регистр (регистры). Кроме того, если установлен сигнал считывания данных, то в конце цикла после загрузки регистра MAR начинает работать память. Данные из памяти помещаются в MBR или MDR в конце *следующего* цикла, а использоваться эти данные могут в цикле, который идет *после* него. Другими словами, если считывание из памяти через любой из портов начинается в конце цикла k, то полученные данные не смогут использоваться в цикле $k + 1$ (только в цикле $k + 2$ и позже).

Это поведение, которое на первый взгляд кажется противоестественным, изображено на рис. 4.2. Сигналы управления памятью выдаются только после загрузки регистров MAR и PC, которая происходит на фронте синхронизирующего сигнала незадолго до конца цикла 1. Будем считать, что память помещает результаты на шину памяти в течение одного цикла, поэтому регистры MBR и/или MDR могут загружаться на следующем фронте вместе с другими регистрами.

Другими словами, мы загружаем регистр MAR в конце цикла тракта данных и инициируем работу памяти сразу после этого. Следовательно, мы не можем ожидать, что результаты считывания окажутся в регистре MDR в начале следующего цикла, особенно если длительность импульса небольшая. Этого времени будет недостаточно. Поэтому между началом считывания из памяти и использованием полученного результата должен помещаться один цикл. Конечно, во время этого цикла могут выполняться и другие операции — не только те, которым требуется слово из памяти.

Предположение о том, что память работает в течение одного цикла, эквивалентно предположению, что доля кэш-попаданий (успешных обращений к кэш-памяти) составляет 100 %. Подобное предположение никогда не может быть истинным, но мы не будем здесь рассказывать о циклах памяти переменной длины, поскольку это не относится к теме книги.

Так как регистры MBR и MDR загружаются на фронте синхронизирующего сигнала вместе с другими регистрами, их можно считывать во время циклов, в течение которых осуществляется передача нового слова из памяти. Они возвращают старые значения, поскольку прошло еще недостаточно времени для того, чтобы они сменились новыми. Здесь нет никакой двусмысленности: до тех пор пока новые значения не загрузятся в регистры MBR и MDR на фронте сигнала, предыдущие значения находятся там и могут использоваться. Отметим, что операции считывания могут проходить одна за другой, то есть в двух последовательных циклах (поскольку сам процесс считывания занимает только один цикл). Кроме того, обе памяти могут функционировать в одно и то же время. Однако попытка чтения и записи одного и того же байта одновременно приводит к неопределенным результатам.

Выходной сигнал шины C можно записать сразу в несколько регистров, однако нежелательно передавать значения более одного регистра на шину B (более того, в некоторых реальных реализациях это приведет к физическому повреждению оборудования). Немного усовершенствовав схемотехнику, мы можем сократить количество битов, необходимых для выбора одного из возможных источников для запуска шины B. Существуют только 9 входных регистров, которые могут запустить шину B (регистры MBR со знаком и без знака учитываются отдельно). Следовательно, мы можем закодировать информацию для шины B в 4 бита и использовать декодер для порождения 16 сигналов управления, 7 из

которых не нужны (у разработчиков коммерческих моделей, возможно, возникло бы желание избавиться от одного из регистров, чтобы обойтись тремя битами, но мы, как ученые, предпочитаем иметь один лишний бит, но при этом получить более понятную конструкцию).

Теперь мы можем управлять трактом данных с помощью 9 + 4 + 8 + 2 + 1 = 24 сигналов, следовательно, нам требуется 24 бита. Однако эти 24 бита управляют трактом данных только в течение одного цикла. Задача управления — определить, что нужно делать в следующем цикле. Чтобы учесть это в конструкции контроллера, мы создадим формат для описания операций, выполняемых с использованием 24 бит управления и двух дополнительных полей: поле NEXT_ADDRESS (следующий адрес) и поле JAM. Содержание каждого из этих полей мы обсудим позже. На рис. 4.4 изображен один из возможных форматов. В нем представлены следующие 6 групп, содержащие 36 сигналов:

- Addr — адрес следующей потенциальной микрокоманды;
- JAM — определение того, как выбирается следующая микрокоманда;
- ALU — функции АЛУ и схемы сдвига;
- C — выбор регистров, которые записываются с шины C;
- Mem — функции памяти;
- B — выбор источника для шины B (как он кодируется, было показано ранее).

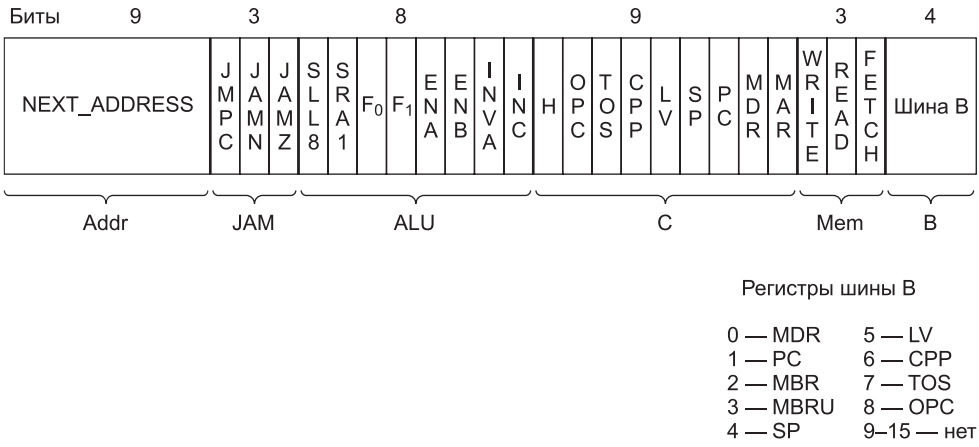


Рис. 4.4. Формат микрокоманды для Mic-1

Порядок групп в принципе произволен, хотя мы долго и тщательно его подбирали, чтобы избежать пересечений на диаграмме, показанной на рис. 4.5. Подобные пересечения на диаграммах часто соответствуют пересечениям проводов на микросхемах. Они значительно затрудняют разработку и их лучше сводить к минимуму.

Управление микрокомандами — микроархитектура Mic-1

До сих пор мы рассказывали об управлении трактом данных и не касались вопроса о том, какой именно сигнал управления и на каком цикле должен уста-

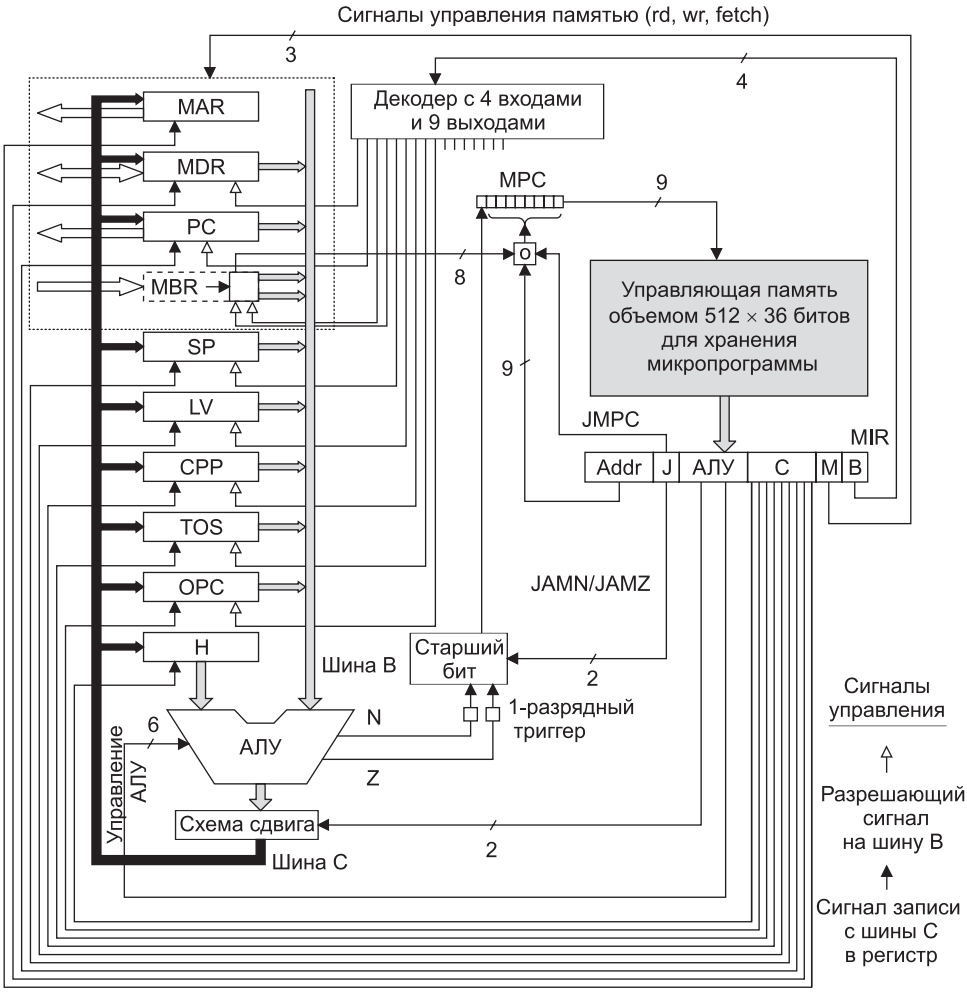


Рис. 4.5. Полная диаграмма микроархитектуры Mic-1

навливаться. Для этого существует **контроллер последовательности**, который отвечает за последовательность операций, необходимых для выполнения одной команды.

Контроллер последовательности в каждом цикле должен выдавать следующую информацию:

- состояние каждого сигнала управления в системе;
- адрес микрокоманды, которая будет выполняться следующей.

Рисунок 4.5 представляет собой подробную диаграмму полной микроархитектуры нашей машины, которую мы назвали **Mic-1**. Хотя на первый взгляд она может показаться внушительной, ее нужно подробно изучить. Если вы разберетесь во всех блоках и их связях, изображенных на этом рисунке, вам легче будет понять структуру уровня микроархитектуры. Диаграмма состоит из двух частей:

тракта данных (*слева*), который мы уже подробно обсудили, и блока управления (*справа*), который мы рассмотрим сейчас.

Самой большой и самой важной частью блока управления является **управляющая память**. Удобно рассматривать ее как память, в которой хранится вся микропрограмма, хотя иногда микропрограмма реализуется в виде набора логических вентилей. Мы будем называть ее управляющей памятью, чтобы не путать с основной памятью, доступ к которой осуществляется через регистры MBR и MDR. Функционально управляющая память представляет собой память, в которой вместо обычных команд хранятся микрокоманды. В нашем примере она содержит 512 слов; каждое слово состоит из одной 36-разрядной микрокоманды, формат которой показан на рис. 4.4. В действительности не все эти слова нужны, но по ряду причин нам требуются адреса для 512 отдельных слов.

Управляющая память отличается от основной памяти тем, что команды, хранящиеся в основной памяти, всегда выполняются в порядке адресов (за исключением переходов), а микрокоманды — нет. Увеличение счетчика команд в листинге 2.1 означает, что команда, которая будет выполняться после текущей, располагается вслед за ней в памяти. Микропрограммы должны обладать большей гибкостью (поскольку последовательности микрокоманд обычно короткие) и этим свойством не обладают. Вместо этого каждая микрокоманда сама указывает на следующую микрокоманду.

Поскольку управляющая память функционально представляет собой ПЗУ, ей нужны собственные адресный регистр и регистр данных. Ей не требуются сигналы чтения и записи, поскольку процесс считывания происходит постоянно. Мы назовем адресный регистр управляющей памяти **MPC** (MicroProgram Counter — **счетчик микропрограмм**). Название не очень подходящее, поскольку микропрограммы не упорядочены явным образом, и понятие счетчика тут неуместно, но мы не можем пойти против традиций. Регистр данных мы назовем **MIR** (MicroInstruction Register — **регистр микрокоманд**). Он содержит текущую микрокоманду, биты которой запускают сигналы управления, влияющие на работу тракта данных.

Регистр MIR, изображенный на рис. 4.5, содержит те же шесть групп сигналов, которые показаны на рис. 4.4. Группы Addr и J (то же, что JAM) контролируют выбор следующей микрокоманды. Мы обсудим их чуть позже. Группа ALU содержит 8 бит, которые позволяют выбрать функцию АЛУ и запустить схему сдвига. Биты C загружают отдельные регистры с шины C. Сигналы M управляют работой памяти.

Наконец, последние 4 бита запускают декодер, который определяет, значение какого регистра будет передано на шину B. В данном случае мы выбрали декодер с четырьмя входами и шестнадцатью выходами, хотя имеется всего 9 разных регистров. В более проработанной модели мог бы использоваться декодер, имеющий 4 входа и 9 выходов. Мы задействуем стандартную схему, чтобы не разрабатывать собственную. Использование стандартных схем упрощает работу, и, кроме того, снижает вероятность ошибок. Ваша собственная микросхема займет меньше места, но на ее разработку потребуется довольно длительное время, к тому же вы можете построить ее неправильно.

Схема, изображенная на рис. 4.5, работает следующим образом. В начале каждого цикла (спад синхронизирующего сигнала на рис. 4.2) в регистр MIR загружа-

ется слово из управляющей памяти, которая на рисунке отмечена буквами MPC. Загрузка регистра MIR занимает период Δw, то есть первый подцикл (см. рис. 4.2).

Когда микрокоманда попадает в MIR, в тракт данных поступают различные сигналы. Значение определенного регистра помещается на шину B, а АЛУ узнает, какую операцию нужно выполнять. Все это происходит во время второго подцикла. После периода $\Delta w + \Delta x$ входные сигналы АЛУ стабилизируются.

После периода Δy стабилизируются сигналы N и Z АЛУ, а также выходной сигнал схемы сдвига. Затем значения N и Z сохраняются в двух 1-разрядных триггерах. Эти биты, как и биты всех регистров, которые загружаются с шины C и памяти, сохраняются на фронте синхронизирующего сигнала, ближе к концу цикла тракта данных. Выходной сигнал АЛУ не сохраняется, а просто передается в схему сдвига. Работа АЛУ и схемы сдвига происходит во время подцикла 3.

После следующего интервала, Δz, выходной сигнал схемы сдвига, пройдя через шину C, достигает регистров. Регистры загружаются в конце цикла на фронте синхронизирующего сигнала (см. рис. 4.2). Во время подцикла 4 происходит загрузка регистров и триггеров N и Z. Подцикл завершается сразу после окончания фронта, когда все значения сохранены, результаты предыдущих операций памяти доступны, регистр MPC загружен. Этот процесс продолжается снова и снова, пока вы не выключите компьютер.

Микропрограмме приходится не только управлять трактом данных, но и определять, какая микрокоманда должна выполняться следующей, поскольку микропрограммы не упорядочены в управляющей памяти. Вычисление адреса следующей микрокоманды начинается после загрузки регистра MIR. Сначала в регистр MPC копируется 9-разрядное поле NEXT_ADDRESS (следующий адрес). Пока происходит копирование, проверяется поле JAM. Если оно содержит значение 000, то ничего больше делать не нужно, и когда копирование поля NEXT_ADDRESS завершится, регистр MPC укажет на следующую микрокоманду.

Если один или несколько битов в поле JAM равны 1, то потребуются еще некоторые действия. Если бит JAMN равен 1, то триггер N соединяется через схему ИЛИ со старшим битом регистра MPC. Если бит JAMZ равен 1, то триггер Z соединяется через схему ИЛИ со старшим битом регистра MPC. Если оба бита равны 1, они оба соединяются через схему ИЛИ с тем же битом. А теперь объясним, зачем нужны триггеры N и Z. Дело в том, что после фронта сигнала (и вплоть до спада) шина B больше не запускается, поэтому выходные сигналы АЛУ уже не могут считаться правильными. Сохранение флагов состояния АЛУ в регистрах N и Z делает правильные значения установившимися и доступными для вычисления регистра MPC, независимо от того, что происходит вокруг АЛУ.

На рис. 4.5 блок, который выполняет это вычисление, помечен как «Старший бит». Он вычисляет следующую булеву функцию:

$$F = ((JAMZ \text{ И } Z) \text{ ИЛИ } (JAMN \text{ И } N)) \text{ ИЛИ } NEXT\_ADDRESS \text{ [8]}$$

Отметим, что в любом случае регистр MPC может принять только одно из двух возможных значений:

1. Значение NEXT_ADDRESS.
2. Значение NEXT_ADDRESS со старшим битом, соединенным операцией ИЛИ с логической единицей.

Других значений не существует. Если старший бит значения NEXT_ADDRESS уже равен 1, нет смысла использовать JAMN или JAMZ.

Отметим, что если все биты JAM равны 0, то адрес следующей команды — это просто 9-разрядное число в поле NEXT_ADDRESS. Если бит JAMN или JAMZ равен 1, то существует два потенциально возможных адреса следующей микрокоманды (символы 0x говорят о том, что следующее за ними число дается в шестнадцатеричной системе счисления): адрес NEXT_ADDRESS и адрес NEXT_ADDRESS, соединенный операцией ИЛИ со значением 0x100 (предполагается, что NEXT_ADDRESS ≤ 0xFF). Рисунок 4.6 поясняет этот момент. Текущая микрокоманда с адресом 0x75 содержит поле NEXT_ADDRESS = 0x92, причем бит JAMZ установлен в 1. Следовательно, следующий адрес микрокоманды зависит от значения бита Z, сохраненного при предыдущей операции АЛУ. Если бит Z равен 0, то следующая микрокоманда имеет адрес 0x92. Если бит Z равен 1, то следующая микрокоманда имеет адрес 0x192.

Рис. 4.6. Микрокоманда с битом JAMZ, равным 1, указывает на две потенциальные последующие микрокоманды

Третий бит в поле JAM — JMPC. Если он установлен, то 8 бит регистра MBR поразрядно связываются операцией ИЛИ с 8 младшими битами поля NEXT_ADDRESS текущей микрокоманды. Результат отправляется в регистр MPC. На рис. 4.5 меткой «O» обозначена схема, которая выполняет операцию ИЛИ над MBR и NEXT_ADDRESS, если бит JMPC равен 1, и просто отправляет NEXT_ADDRESS в регистр MPC, если бит JMPC равен 0. Если бит JMPC равен 1, то младшие 8 бит поля NEXT_ADDRESS равны 0. Старший бит может быть 0 или 1, поэтому значение поля NEXT_ADDRESS обычно 0x000 или 0x100. Почему иногда используется значение 0x000, а иногда — 0x100, мы обсудим позже.

Возможность выполнения операции ИЛИ над MBR и NEXT_ADDRESS и сохранения результата в регистре MPC позволяет реализовывать межуровневые переходы. Отметим, что биты, находящиеся в регистре MBR, позволяют задать любой адрес из 256 возможных. Регистр MBR содержит код операции, поэтому использование бита JMPC приведет к единственно возможному выбору следующей микрокоманды. Этот метод позволяет осуществлять быстрый переход к функции, соответствующей вызванному коду операции.

Чтобы разобраться в следующем материале этой главы, очень важно понимать принципы синхронизации машины, поэтому повторим их еще раз. Синхронизирующий сигнал делится на подциклы, хотя внешние изменения этого

сигнала происходят только на спаде, с которого начинается цикл, и на фронте, который загружает регистры и триггеры N и Z. Посмотрите еще раз на рис. 4.2.

Во время подцикла 1, который инициируется спадом сигнала, адрес, находящийся в регистре MPC, загружается в регистр MIR. Во время подцикла 2 регистр MIR устанавливает сигналы, и на шину B загружается выбранный регистр. Во время подцикла 3 работают АЛУ и схема сдвига. Во время подцикла 4 стабилизируются значения шины C, шин памяти и АЛУ. На фронте сигнала загружаются регистры из шины C, загружаются триггеры N и Z, а регистры MBR и MDR получают результаты из памяти, начавшей функционировать в конце предыдущего цикла (если эти результаты вообще имеются). Как только регистр MBR получает свое значение, загружается регистр MPC. Это происходит где-то в середине отрезка между фронтом и спадом, но уже после загрузки регистров MBR и MDR. Регистр MPC может загружаться либо уровнем (но не фронтом) сигнала, либо через фиксированный отрезок времени после фронта. Все это означает, что регистр MPC не получает своего значения до тех пор, пока не будут готовы регистры MBR, N и Z, от которых он зависит. На спаде сигнала, когда начинается новый цикл, регистр MPC может обращаться к памяти.

Отметим, что каждый цикл является самодостаточным. В каждом цикле определяется, значение какого регистра должно поступать на шину B, что должны делать АЛУ и схема сдвига, куда нужно сохранить значение шины C и, наконец, каким должно быть следующее значение регистра MPC.

Следует сделать еще одно замечание по поводу рис. 4.5. До сих пор мы считали MPC регистром, который состоит из 9 бит и загружается на высоком уровне сигнала. В действительности этот регистр вообще не нужен. Все его входные сигналы можно непосредственно связать с управляющей памятью. Поскольку эти сигналы присутствуют в управляющей памяти на спаде синхронизирующего сигнала, когда выбирается и считывается регистр MIR, этого достаточно. Их не нужно хранить в регистре MPC. По этой причине MPC может быть реализован в виде **виртуального регистра**, который представляет собой просто место объединения сигналов и похож скорее на коммутационное поле, чем на настоящий регистр. Если MPC сделать виртуальным регистром, то процедура синхронизации значительно упрощается: в этом случае события происходят только на фронте и спаде сигнала. Но если вам проще считать MPC реальным регистром, то такой подход тоже вполне допустим.

Пример архитектуры набора команд — IJVM

Чтобы продолжить изучение нашего примера, познакомимся с уровенем архитектуры набора команд (ISA), которые должна интерпретировать микропрограмма машины IJVM (см. рис. 4.5). Для удобства уровень архитектуры команд мы иногда будем называть **макроархитектурой**, чтобы противопоставить его микроархитектуре. Однако перед тем как приступать к описанию IJVM, мы немного отвлечемся.

Стек

Практически в любом языке программирования существует понятие процедур (методов), имеющих локальные переменные. Эти переменные доступны во время

выполнения процедуры, но перестают быть доступными после ее окончания. Возникает вопрос: где должны храниться такие переменные?

Простейшее решение — связать каждую переменную с абсолютным адресом в памяти — не работает. Проблема заключается в том, что процедура может вызывать себя сама. Мы рассмотрим такие рекурсивные процедуры в главе 5. А пока достаточно сказать, что если процедура вызывается дважды, то хранить ее переменные под конкретными адресами в памяти нельзя, поскольку второй вызов будет конфликтовать с первым.

Вместо этого используется другая стратегия. Для переменных резервируется особая область памяти, которая называется **стеком** и в которой отдельные переменные не получают абсолютных адресов. Какой-либо регистр, скажем, LV, указывает на базовый адрес локальных переменных для текущей процедуры. Посмотрите на рис. 4.7, *а*. В данном случае вызывается процедура *A* с локальными переменными *a*1, *a*2 и *a*3, и для этих переменных резервируется участок памяти, начинающийся с адреса, на который указывает регистр LV. Другой регистр, SP, указывает на старшее слово локальных переменных процедуры *A*. Если значение регистра LV равно 100, а размер слова составляет 4 байта, то значение SP составляет 108. Для обращения к переменной нужно вычислить ее смещение от адреса LV. Структура данных между LV и SP (включая оба указанных слова) называется **кадром локальных переменных**.

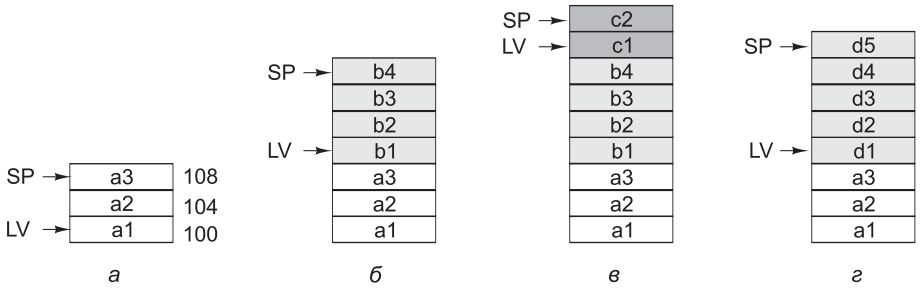


Рис. 4.7. Стек для хранения локальных переменных: во время вызова процедуры A (*а*); после того как процедура A вызывает процедуру B (*б*); после того как процедура B вызывает процедуру C (*в*); после того как процедуры C и B завершаются, а процедура A вызывает процедуру D (*г*)

А теперь давайте посмотрим, что произойдет, если процедура *A* вызывает другую процедуру, например *B*. Где должны храниться 4 локальные переменные процедуры *B* (*b*1, *b*2, *b*3, *b*4)? Ответ: в стеке, расположенном над стеком для процедуры *A*, как показано на рис. 4.7, *б*. Отметим, что после вызова процедуры регистр LV указывает уже на локальные переменные процедуры *B*. Обращаться к локальным переменным процедуры *B* можно по их сдвигу от LV. Если процедура *B* вызывает процедуру *C*, то регистры LV и SP снова переопределяются и указывают на местонахождение локальных переменных процедуры *C*, как показано на рис. 4.7, *в*.

Когда процедура *C* завершается, *B* снова активизируется, и стек возвращается в прежнее состояние (см. рис. 4.7, *б*), так что LV теперь указывает на локальные переменные процедуры *B*. Когда процедура *B* завершается, стек возвращается в исходное состояние (см. рис. 4.7, *а*). LV всегда указывает на базовый адрес кадра локальных переменных текущей процедуры, а SP — на верхнее слово этого кадра.

Предположим, что процедура *A* вызывает процедуру *D*, которая содержит 5 локальных переменных. Соответствующий стек показан на рис. 4.7, *г*. Локальные переменные процедуры *D* используют область памяти процедуры *B* и часть стека процедуры *C*. В памяти с такой организацией размещаются только текущие процедуры. Когда процедура завершается, отведенный для нее участок памяти освобождается.

Но стек используется не только для хранения локальных переменных, но и для хранения операндов во время вычисления арифметических выражений. Такой стек называется **стеком операндов**. Предположим, что перед вызовом процедуры *B* процедура *A* должна произвести следующее вычисление:

$$a1 = a2 + a3.$$

Чтобы вычислить эту сумму, можно поместить *a*2 в стек, как показано на рис. 4.8, *а*. Тогда значение регистра SP увеличится на число, равное количеству байтов в слове (скажем, на 4), и будет указывать на адрес первого операнда. Затем в стек помещается переменная *a*3, как показано на рис. 4.8, *б*. (Далее в тексте имена переменных и процедур будут выделяться курсивом.)

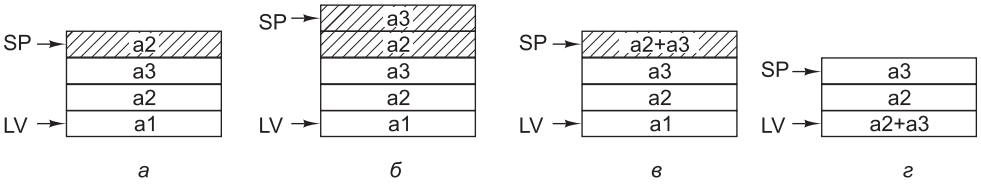


Рис. 4.8. Использование стека операндов для выполнения арифметических действий

Теперь можно произвести вычисления, выполнив команду, которая выталкивает два слова из стека, складывает их и помещает результат обратно в стек, как показано на рис. 4.8, *в*. После этого верхнее слово можно вытолкнуть из стека и поместить его в локальную переменную *a*1, как показано на рис. 4.8, *г*.

Кадры локальных переменных и стеки операндов могут смешиваться. Например, когда вызывается функция *f* при вычислении выражения $x^2 + f(x)$, часть этого выражения (x^2) может находиться в стеке операндов. Результат вычисления функции остается в стеке над x^2, чтобы следующая команда могла сложить операнды.

Стоит заметить, что все машины используют стек для хранения локальных переменных, но не все используют его для хранения операндов. В большинстве машин нет стека операндов, но и у JVM, и у IJVM он есть. Стековые операции мы рассмотрим подробно в главе 5.

Модель памяти IJVM

Теперь можно переходить к рассмотрению архитектуры IJVM. Она состоит из памяти, которую можно рассматривать либо как массив из 4 294 967 296 байт (4 Гбайт), либо как массив из 1 073 741 824 слов, каждое из которых содержит 4 байта. В отличие от большинства архитектур команд, виртуальная машина Java не выполняет обращений к памяти, видимых на уровне команд, но имеет несколько неявных адресов, которые составляют основу указателя. IJVM-команды могут обращаться к памяти только через эти указатели. Определены следующие области памяти:

Набор констант. Эта область, недоступная для записи из IJVM-программы, состоит из констант, строк и указателей на другие области памяти, на которые можно делать ссылку. Данная область загружается в момент загрузки программы в память и после этого не меняется. Имеется скрытый регистр CPP (Constant Pool Pointer — указатель набора констант), который содержит адрес первого слова набора констант.

Кадр локальных переменных. Эта область предназначена для хранения переменных во время выполнения процедуры. Она, как уже отмечалось, называется **кадром локальных переменных**. В начале этого кадра располагаются параметры (или аргументы) вызванной процедуры. Кадр локальных переменных не включает в себя стек операндов, который размещается отдельно. По соображениям производительности мы поместили стек операндов прямо над кадром локальных переменных. Существует скрытый регистр, который содержит адрес первой переменной кадра. Мы назовем этот регистр LV (Local Variable — локальная переменная). Параметры вызванной процедуры хранятся в начале кадра локальных переменных.

Стек операндов. Стек операндов не должен быть больше определенного размера, который заранее вычисляется компилятором Java. Пространство стека операндов располагается прямо над кадром локальных переменных, как показано на рис. 4.9. В данном случае стек операндов удобно считать частью кадра локальных переменных. В любом случае существует виртуальный регистр, который содержит адрес верхнего слова стека. Отметим, что в отличие от регистров CPP и LV, этот указатель меняется во время выполнения процедуры, поскольку операнды помещаются в стек и выталкиваются из него.

Область процедур. Наконец, существует область памяти, в которой содержится программа. Скрытый регистр содержит адрес команды, которая должна вызываться следующей. Этот указатель называется счетчиком команд (Program Counter, PC). В отличие от других областей памяти, область процедур представляет собой массив байтов.

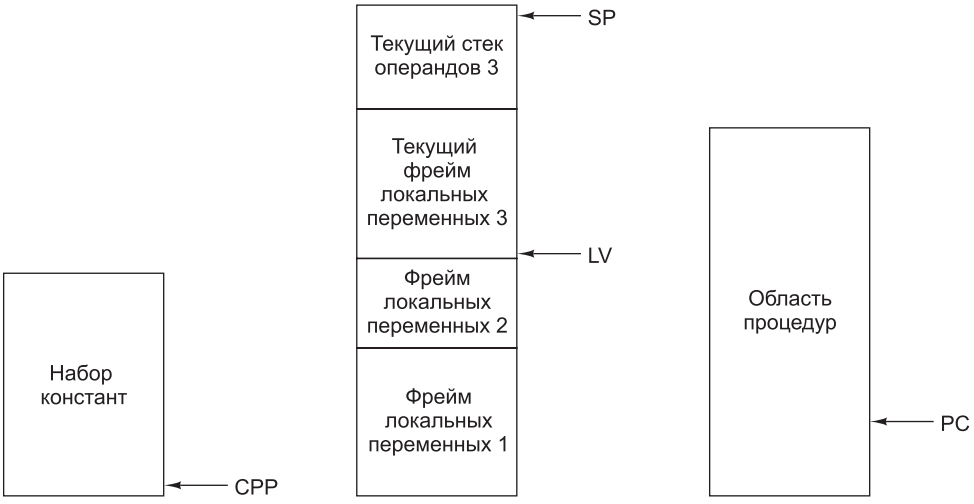


Рис. 4.9. Области памяти IJVM

Следует сделать одно замечание по поводу указателей. Регистры CPP, LV и SP указывают на *слова*, а не на *байты*, и смещения происходят на определенное число слов. Например, значения LV, LV + 1 и LV + 2 указывают на первые три слова кадра локальных переменных, а LV, LV + 4 и LV + 8 — на слова, расположенные на расстоянии четырех слов (16 байт) друг от друга.

Регистр PC, напротив, содержит адреса байтов, и его изменение означает увеличение на определенное количество байтов, а не слов. Обращение к памяти регистра PC отличается от обращений других регистров, поэтому в машине Mic-1 и предусмотрен специальный порт памяти для PC. Его размер составляет всего один байт. Если увеличить PC на единицу и выполнить чтение, это приведет к вызову следующего *байта*. А если увеличить SP на единицу и выполнить чтение, это приведет к вызову следующего *слова*.

Набор IJVM-команд

Набор IJVM-команд приведен в табл. 4.2. Каждая команда состоит из кода операции и иногда из операнда (например, смещения адреса или константы). В первом столбце приводится шестнадцатеричный код команды. Во втором столбце дается мнемоника на языке ассемблера. В третьем столбце описывается назначение команды.

Таблица 4.2. Набор IJVM-команд. Размер операндов byte, const и varnum — 1 байт. Размер операндов disp, index и offset — 2 байта

Hex	Мнемоника	Примечание
0x10	BIPUSH *byte*	Помещает байт в стек
0x59	DUP	Копирует верхнее слово стека и помещает его в стек
0xA7	GOTO *offset*	Безусловный переход
0x60	IADD	Выталкивает два слова из стека; помещает в стек их сумму
0x7E	IAND	Выталкивает два слова из стека; помещает в стек результат логического умножения (операция И)
0x99	IFEQ *offset*	Выталкивает слово из стека и совершает переход, если оно равно нулю
0x9B	IFLT *offset*	Выталкивает слово из стека и совершает переход, если оно меньше нуля
0x9F	IF_ICMPEQ *offset*	Выталкивает два слова из стека; совершает переход, если они равны
0x84	IINC *varnum const*	Прибавляет константу к локальной переменной
0x15	ILOAD *varnum*	Помещает локальную переменную в стек
0xB6	INVOKEVIRTUAL *disp*	Вызывает процедуру
0x80	IOR	Выталкивает два слова из стека; помещает в стек результат логического сложения (операция ИЛИ)

Hex	Мнемоника	Примечание
0xAC	IRETURN	Выдает результат выполнения процедуры (целое число)
0x36	ISTORE *varnum*	Выталкивает слово из стека и запоминает его в кадре локальных переменных
0x64	ISUB	Выталкивает два слова из стека; помещает в стек их разность
0x13	LDC_W *index*	Берет константу из набора констант и помещает ее в стек
0x00	NOP	Не производит никаких действий
0x57	POP	Удаляет верхнее слово стека
0x5F	SWAP	Переставляет два верхних слова стека
0xC4	WIDE	Префиксная команда; следующая команда содержит 16-разрядный индекс

Некоторые команды помещают слова из различных источников в стек. Такими источниками могут быть набор констант (`LDC_W`), кадр локальных переменных (`ILOAD`) и сама команда (`BIPUSH`). Переменную можно также вытолкнуть из стека и сохранить ее в кадре локальных переменных (`ISTORE`). Над двумя верхними словами стека можно совершать две арифметические (`IADD` и `ISUB`) и две логические операции (`IAND` и `IOR`). При выполнении любой арифметической или логической операции два слова выталкиваются из стека, а результат помещается обратно в стек. Существуют 4 команды перехода: одна для безусловного перехода (`GOTO`), а три другие для условных переходов (`IFEQ`, `IFLT` и `IF_ICMPEQ`). Все эти команды изменяют значение PC на размер их смещения, который следует за кодом операции в команде. Операнд смещения состоит из 16 бит. Он прибавляется к адресу кода операции. Существуют также команды для перестановки двух верхних слов стека (`SWAP`), дублирования верхнего слова (`DUP`) и удаления верхнего слова (`POP`).

Некоторые команды имеют сложный формат, допускающий краткую форму записи для часто используемых версий. Из всех механизмов, которые JVM применяет для этого, в IJVM мы включили два. В одном случае мы пропустили краткую форму в пользу более традиционной. В другом случае мы показываем, как префиксная команда `WIDE` может использоваться для изменения следующей команды.

Наконец, существуют команда вызова другой процедуры (`INVOKEVIRTUAL`) и команда выхода из текущей процедуры и возвращения к процедуре, из которой она была вызвана (`IRETURN`). Из-за сложности механизма мы немного упростили определение. Ограничение состоит в том, что, в отличие от языка Java, в нашем примере процедура может вызывать только такую процедуру, которая находится внутри нее. Хотя это ограничение противоречит сути языка Java, оно позволяет представить более простой механизм без необходимости размещать процедуру динамически. (Если вы не знакомы с объектно-ориентированным программированием, можете игнорировать это предложение. Мы просто превратили язык Java из объектно-ориентированного в обычный, такой как C или Pascal.)

На всех компьютерах, кроме JVM, адрес процедуры, которую нужно вызвать, непосредственно определяется командой CALL, поэтому наш подход скорее правило, чем исключение.

Механизм вызова процедуры состоит в следующем. Сначала вызывающая программа помещает в стек указатель на вызываемый объект. На рис. 4.10, *а* этот указатель обозначен символами OBJREF. Затем вызывающая программа помещает в стек параметры процедуры (в данном примере — это *Параметр 1*, *Параметр 2* и *Параметр 3*). После этого выполняется команда INVOKEVIRTUAL.

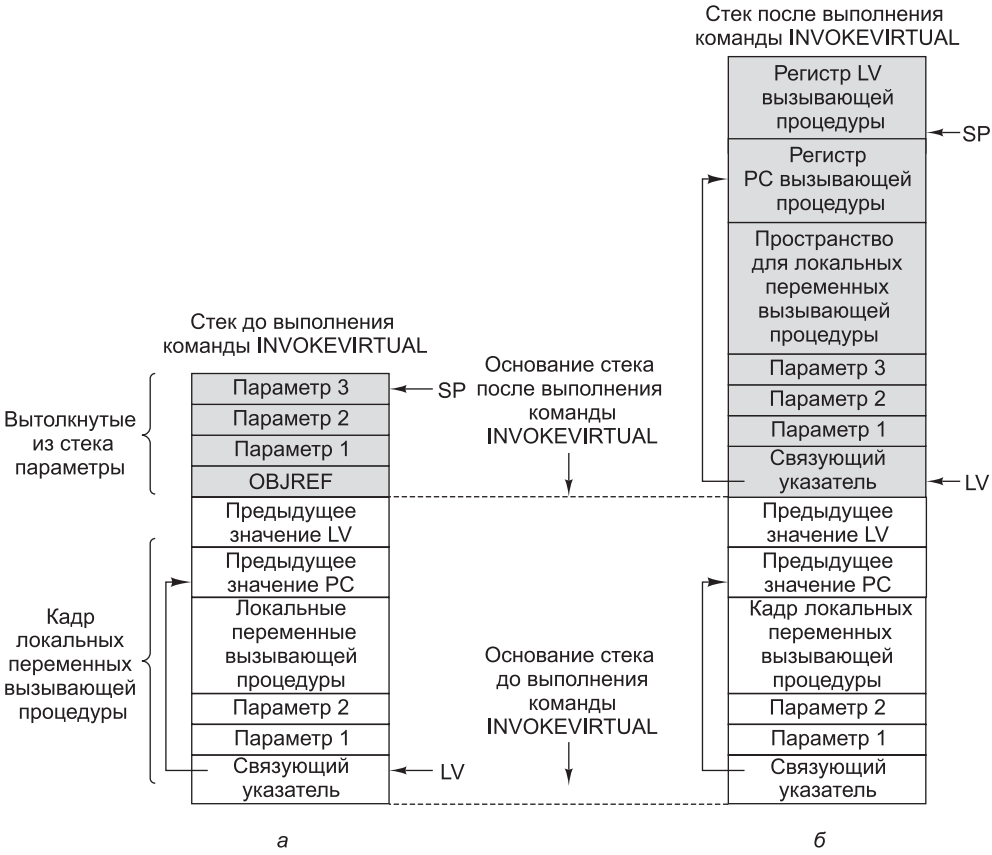


Рис. 4.10. Память до выполнения команды INVOKEVIRTUAL (*а*); память после выполнения этой команды (*б*)

Команда INVOKEVIRTUAL включает в себя смещение, которое определяет позицию в наборе констант. В этой позиции находится начальный адрес вызываемой процедуры, которая хранится в области процедур. Первые 4 байта в области процедур содержат специальные данные. Первые 2 байта представляют собой целое 16-разрядное число, указывающее на количество параметров данной процедуры (сами параметры были ранее помещены в стек). В данном случае указатель OBJREF считается параметром — параметром 0. Это 16-разрядное целое число вместе со значением SP дает адрес OBJREF. Отметим, что регистр LV указывает

на OBJREF, а не на первый реальный параметр. Выбор того, на что указывает LV, в какой-то степени произволен.

Следующие 2 байта в области процедур представляют еще одно 16-разрядное целое число, задающее размер области локальных переменных для вызываемой процедуры. Дело в том, что для данной процедуры предоставляется новый стек, который размещается прямо над кадром локальных переменных, для этого и нужно это число. Наконец, пятый байт в области процедур содержит код первой операции, которую нужно выполнить.

Посмотрим, что происходит перед вызовом процедуры (см. также рис. 4.10). Два байта без знака, которые следуют за кодом операции, используются для индексирования таблицы констант (первый байт — старший). Команда вычисляет базовый адрес нового кадра локальных переменных. Для этого из указателя стека вычитается число параметров, а LV устанавливается на OBJREF. В OBJREF хранится адрес ячейки, в которой находится старое значение PC. Этот адрес вычисляется суммированием размера кадра локальных переменных (параметры + локальные переменные) с адресом, содержащимся в регистре LV. Сразу над адресом, предназначенным для сохранения старого значения PC, находится адрес, в котором должно быть сохранено старое значение LV. Над этим адресом начинается стек для новой вызванной процедуры. SP указывает на старое значение LV, адрес которого находится сразу под первой пустой ячейкой стека. Напомним, что SP всегда указывает на верхнее слово в стеке. Если стек пуст, то SP указывает на адрес, который находится непосредственно под стеком. На наших рисунках стек всегда заполняется снизу вверх, по направлению к старшим адресам у верхнего края страницы.

И наконец, для выполнения команды `INVOKEVIRTUAL` регистр PC должен указывать на пятый байт в кодовом пространстве процедуры.

Команда `IRETURN` противоположна команде `INVOKEVIRTUAL` (рис. 4.11). Она освобождает память, используемую процедурой, а также возвращает стек в предыдущее состояние, за исключением того, что, во-первых, OBJREF и все параметры удаляются из стека; во-вторых, возвращенное значение помещается в стек, туда, где раньше находился параметр OBJREF. Чтобы восстановить прежнее состояние, команда `IRETURN` должна вернуть прежние значения указателей PC и LV. Для этого она обращается к связующему указателю (это слово, определяемое текущим значением LV). В этом месте, где изначально находился параметр OBJREF, команда `INVOKEVIRTUAL` сохранила адрес, содержащий старое значение PC. Это слово, а также слово над ним извлекаются, чтобы восстановить старые значения PC и LV соответственно. Возвращенное значение, которое хранится на самой вершине стека завершающейся процедуры, копируется туда, где изначально находился параметр OBJREF, после чего SP начинает указывать на этот адрес. И тогда управление передается команде, которая следует сразу за `INVOKEVIRTUAL`.

До сих пор у нашей машины не было никаких команд ввода-вывода. Мы и не собираемся их вводить. В нашем примере, как и в виртуальной машине Java, они не нужны, и в описании JVM никогда не упоминаются процессы ввода-вывода. Считается, что машина без механизмов ввода-вывода более надежна. (Чтение и запись осуществляется в JVM путем вызова специальных процедур.)

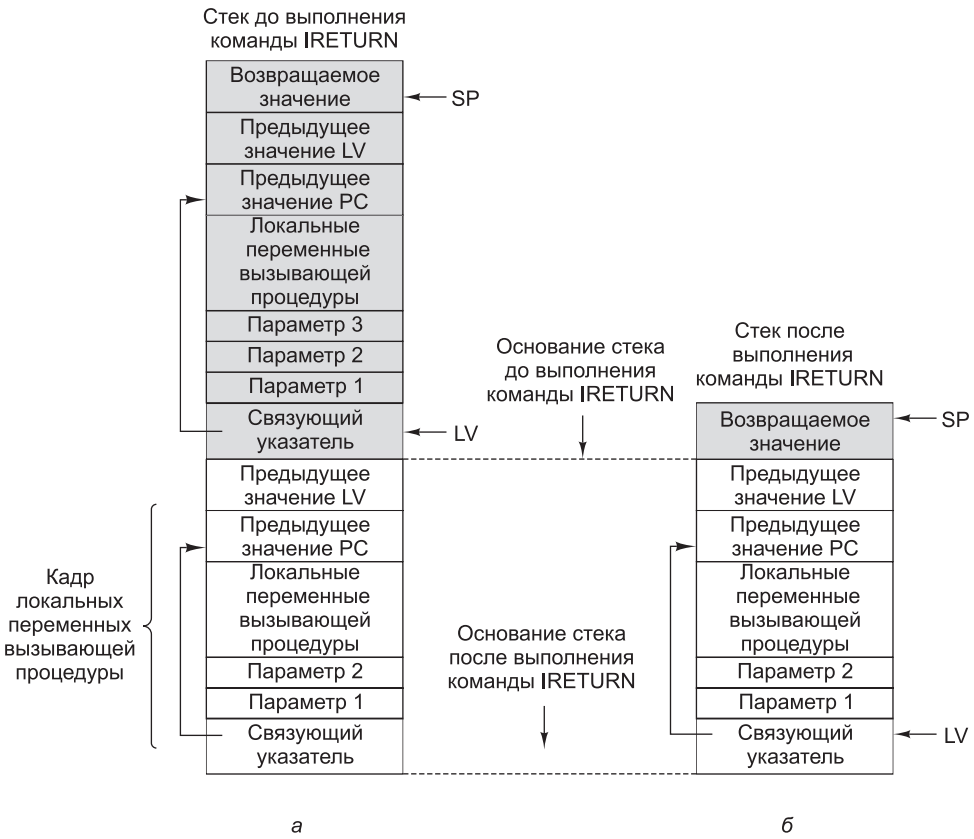


Рис. 4.11. Память до выполнения команды IRETURN (*а*); память после выполнения этой команды (*б*)

Компиляция IJVM

А теперь посмотрим, какое отношение язык Java имеет к IJVM. В листинге 4.1 представлен небольшой фрагмент программы на языке Java. Компилятор Java должен был бы переделать эту программу в программу на языке ассемблера IJVM, приведенную в листинге 4.2. Номера строк 1–15 в левой части листинга, а также комментарии после символов двойной косой черты (//) не являются частью самой программы. Они приведены для наглядности и просто облегчают понимание. Ассемблер Java транслировал бы в программу в двоичный код, показанный в листинге 4.3. (В действительности компилятор Java сразу генерирует двоичный код без промежуточного ассемблирования.) В данном примере `i` — локальная переменная 1, `j` — локальная переменная 2, а `k` — локальная переменная 3.

Листинг 4.1. Фрагмент программы на языке Java

```
i=j+k;
if(i==3)
    k=0;
else
    j=j-1;
```

Листинг 4.2. Программа для IJVM на языке ассемблера Java

```
 1      ILOAD j         // i=j+k
 2      ILOAD k
 3      IADD
 4      ISTORE i
 5      ILOAD i         // if(i==3)
 6      BIPUSH 3
 7      IF_ICMPEQ L1
 8      ILOAD j         // j=j-1
 9      BIPUSH 1
10      ISUB
11      ISTORE j
12      GOTO L2
13  L1: BIPUSH 0       // k=0
14      ISTORE k
15  L2:
```

Листинг 4.3. Программа для IJVM в шестнадцатеричном коде

```
0x15 0x02
0x15 0x03
0x60
0x36 0x01
0x15 0x01
0x10 0x03
0x9F 0x00 0x0D
0x15 0x02
0x10 0x01
0x64
0x36 0x02
0xA7 0x00 0x07
0x10 0x00
0x36 0x03
```

Откомпилированная программа проста. Сначала `j` и `k` помещаются в стек, складываются, а результат сохраняется в `i`. Затем `i` и константа 3 помещаются в стек и сравниваются. Если они равны, то совершается условный переход к `L1`, где `k` получает значение 0. Если они не равны, то выполняется часть программы после команды `IF_ICMPEQ`. После этого осуществляется переход к `L2`, где объединяются части `else` и `then`.

Стек операндов для программы, приведенной в листинге 4.2, изображен на рис. 4.12. До начала выполнения программы стек пуст, что показано горизонтальной чертой над цифрой 0. После выполнения первой команды `ILOAD j` помещается в стек (прямоугольник над цифрой 1 на рисунке). Цифра 1 означает, что выполнена первая команда. После выполнения второй команды `ILOAD` в стеке оказываются уже два слова, как показано в прямоугольнике над цифрой 2. После выполнения команды `IADD` в стеке остается только одно слово, которое представляет собой сумму `j` + `k`. Когда верхнее слово выталкивается из стека и сохраняется в `i`, стек снова становится пустым.

Команда 5 (`ILOAD`) начинает оператор `if`. Эта команда помещает `i` в стек. Затем идет константа 3 (в команде 6). После сравнения стек снова становится пустым (7). Команда 8 является началом фрагмента `else`. Он продолжается вплоть до команды 12, когда совершается переход к метке `L2`.

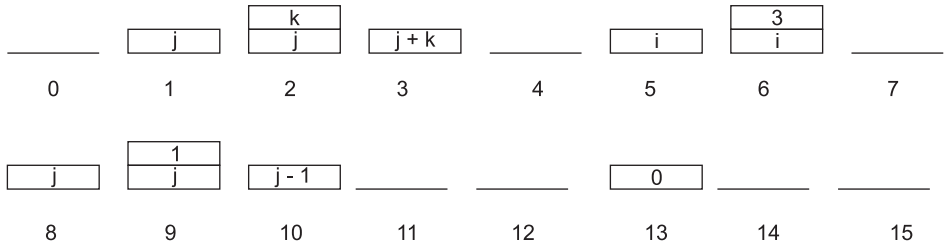


Рис. 4.12. Состояние стека после выполнения каждой команды в программе, приведенной в листинге 4.2

Пример реализации микроархитектуры

Мы подробно описали микро- и макроархитектуру. Осталось осуществить реализацию. Другими словами, нам предстоит узнать, что собой представляет и как работает программа уровня микроархитектуры, интерпретирующая микрокоманды. Прежде чем это сделать, познакомимся с системой обозначений, которую мы будем использовать для описания.

Микрокоманды и их запись

В принципе, мы могли бы описать работу управляющей памяти с помощью двоичной системы счисления, по 36 бит в слове. Но гораздо удобнее ввести систему обозначений, с помощью которой можно передать суть рассматриваемых вопросов и при этом не вдаваться в ненужные подробности. Важно понимать, что язык, который мы выбираем, призван проиллюстрировать основные принципы работы программы, а вовсе не предназначен для использования в новых проектах. Если бы нашей целью было практическое применение языка, мы бы ввели совсем другую запись, чтобы максимально повысить гибкость программы. При этом была бы очень важна проблема выбора адресов, поскольку адреса в памяти неупорядочены. Соответственно эффективность такой организации управления во многом зависит от умения разработчика грамотно выбирать адреса. Поэтому мы введем простой символический язык, который полностью описывает каждую операцию, но без полного определения всех необходимых адресов.

Наша система обозначений показывает все действия, которые происходят на одной линии за один цикл. Теоретически для описания этих операций мы могли бы использовать язык высокого уровня, однако такой язык не позволяет разработчику получить полный контроль над циклами. Благодаря такому контролю можно проанализировать каждый цикл, разобраться во всех операциях и проверить их. Если целью разработки является повышение скорости и производительности, важен каждый цикл. При практической реализации для экономии циклов может использоваться множество различных приемов. В подобной экономии есть большая выгода: если команду из четырех циклов сократить на два цикла, она будет выполняться в два раза быстрее, причем это повышение скорости будет достигаться при каждом выполнении этой команды.

Одно из возможных решений — просто составить список сигналов, которые должны активизироваться в каждом цикле. Предположим, что в одном цикле

мы хотим увеличить значение SP на единицу. Еще мы хотим инициировать операцию чтения, причем так, чтобы следующая команда находилась в ячейке 122 управляющей памяти. Тогда мы могли бы написать:

```
ReadRegister=SP, ALU=INC, WSP, Read, NEXT_ADDRESS=122
```

Здесь WSP значит «записать регистр SP». Эта запись полная, но она сложна для понимания. Вместо этого мы объединим указанные операции и передадим в записи результат действий:

```
SP = SP + 1; rd
```

Назовем наш микроассемблер высокого уровня MAL (Micro Assembly Language — микроассемблер). По-французски «mal» означает «болезнь» — это то, что с вами случится, если вы будете писать слишком большие программы на этом языке. Язык MAL придуман исключительно для демонстрации основных характеристик микроархитектуры. Во время каждого цикла могут записываться любые регистры, но обычно записывается только один. Значение только одного регистра может передаваться на шину B и в АЛУ. На шине A может быть +1, 0, –1 и регистр H. Следовательно, для обозначения определенной операции мы можем использовать простой оператор присваивания, как в языке Java. Например, чтобы копировать регистр SP в регистр MDR, мы можем написать:

```
MDR = SP
```

Чтобы показать, что мы используем какую-либо функцию АЛУ, мы можем написать, например, так:

```
MDR = H + SP
```

Эта строка означает, что значение регистра H складывается со значением регистра SP и результат записывается в регистр MDR. Операция сложения коммутативна (это значит, что порядок операндов не имеет значения), поэтому данное выше выражение можно записать в виде:

```
MDR = SP + H
```

При этом генерируется та же 36-разрядная микрокоманда, хотя, строго говоря, H является левым операндом АЛУ.

Использовать можно только допустимые операции; самые важные из них перечислены ниже:

```
DEST = H
DEST = SOURCE
DEST = H
DEST = SOURCE
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H И SOURCE
DEST = H ИЛИ SOURCE
DEST = 0
DEST = 1
DEST = –1
```

Здесь SOURCE — значение любого из регистров MDR, PC, MBR, MBRU, SP, LV, CPP, TOS и OPC (MBRU — версия MBR без знака). Все эти регистры могут выступать в качестве источников значений для АЛУ (они поступают в АЛУ через шину B). Сходным образом DEST может обозначать любой из следующих регистров: MAR, MDR, PC, SP, LV, CPP, TOS, OPC и H; каждый из этих регистров может быть целевым для выходного сигнала АЛУ, который передается к регистрам по шине C. Многие, казалось бы разумные команды недопустимы. Например, следующее выражение выглядит вполне корректно, но эту операцию нельзя выполнить в тракте данных, изображенном на рис. 4.5, за один цикл:

```
MDR = SP + MDR
```

Такое ограничение существует, поскольку для операции сложения (в отличие от увеличения или уменьшения на 1) один из операндов должен быть значением регистра H. Точно так же могло бы пригодиться, например, такое выражение:

```
H = H – MDR
```

Однако оно недопустимо, поскольку единственным возможным источником вычитаемого является регистр H. Ассемблер должен отклонять выражения, которые кажутся пригодными, но в действительности недопустимы.

В нашей системе записи допускается использование нескольких операторов присваивания. Например, чтобы прибавить 1 к регистру SP и сохранить полученное значение в регистрах SP и MDR, нужно записать следующее:

```
SP = MDR = SP + 1
```

Для обозначения процессов считывания из памяти и записи в память слов по 4 байта мы будем вставлять в микрокоманду операторы `rd` и `wr`. Для получения байта через 1-байтный порт используется оператор `fetch`. Операции присваивания и операции взаимодействия с памятью могут происходить в одном и том же цикле. То, что происходит в одном цикле, записывается в одну строку.

Чтобы избежать путаницы, напомним еще раз, что Mic-1 может обращаться к памяти двумя способами. При чтении и записи 4-байтных слов данных используются регистры MAR и MDR. Эти процессы показываются в микрокомандах операторами `rd` и `wr` соответственно. При чтении 1-байтных кодов операций из потока команд используются регистры PC и MBR. В микрокоманде это показывается оператором `fetch`. Оба типа операций взаимодействия с памятью могут происходить одновременно.

Однако один и тот же регистр не может получать значение из памяти и тракта данных в одном и том же цикле. Рассмотрим фрагмент программы:

```
MAR = SP; rd
MDR = H
```

В результате выполнения первой микрокоманды значение из памяти приписывается регистру MDR в конце второй микрокоманды. Однако вторая микрокоманда в то же самое время приписывает другое значение регистру MDR. Эти две операции присваивания конфликтуют, поскольку результаты не определены.

Помните, что в каждой микрокоманде должен быть явно указан адрес следующей микрокоманды. Однако часто бывает так, что микрокоманда вызывается только одной микрокомандой, причем именно той, которая находится в предыдущей строке. Чтобы упростить работу программиста, микроассемблер обычно приписывает адрес каждой микрокоманде (порядок следования адресов может

и не соответствовать последовательности микрокоманд в управляющей памяти) и заполняет поле NEXT_ADDRESS, так что последовательность выполнения микрокоманд соответствует очередности строк микропрограммы.

Однако программисту иногда нужно совершить переход, условный или безусловный. Запись безусловных переходов проста:

```
goto метка
```

Такая запись может включаться в любую микрокоманду, при этом в ней явным образом указывается имя следующей микрокоманды. Например, очень часто последовательность микрокоманд заканчивается возвращением к первой команде основного цикла, поэтому последняя команда в каждой такой последовательности содержит запись:

```
goto Main1
```

Отметим, что в тракте данных происходят обычные операции даже во время выполнения микрокоманд, которые содержат оператор `goto`. В любой микрокоманде есть поле NEXT_ADDRESS. Оператор `goto` сообщает ассемблеру, что в это поле вместо адреса микрокоманды, записанной в следующей строке, нужно поместить особое значение. В принципе, каждая строка должна содержать запись `goto`, но если нужный адрес — это адрес микрокоманды, записанной в следующей строке, `goto` для удобства можно опускать.

Для условных переходов потребуется другая запись. Вспомните, что JAMN и JAMZ используют биты N и Z соответственно. Например, иногда нужно проверить, не равно ли нулю значение регистра. Для этого можно было бы пропустить это значение через АЛУ, сохранив его после этого в том же регистре. Тогда мы бы написали:

```
TOS = TOS
```

Запись выглядит забавно, но выполняет необходимые действия (устанавливает триггер Z и записывает значение в регистре TOS). В целях удобочитаемости микропрограммы мы расширили язык MAL, добавив два новых воображаемых регистра N и Z, которым можно присваивать значения. Например:

```
Z = TOS
```

Эта строка пропускает значение регистра TOS через АЛУ, устанавливая триггер Z (и N), но при этом не сохраняет значение ни в одном из регистров. Использование регистра Z или N в качестве целевого показывает микроассемблеру, что нужно установить в 0 все биты поля C (см. рис. 4.4). Тракт данных проходит обычный цикл, выполняются все обычные допустимые операции, но не один из регистров не записывается. Не важно, какой регистр является целевым, N или Z. Микрокоманды, которые при этом генерирует микроассемблер, одинаковы. (Программисты, намеренно выбравшие не тот регистр, в наказание будут неделю работать на самом первом компьютере IBM PC с частотой 4,77 МГц.)

Чтобы микроассемблер установил бит JAMZ, нужно использовать следующий синтаксис:

```
if(Z) goto L1; else goto L2
```

Поскольку аппаратное обеспечение требует, чтобы 8 младших битов этих адресов совпадали, микроассемблер должен присвоить им такие адреса. С другой стороны, метка `L2` может находиться в любом из младших 256 слов управляющей памяти, поэтому микроассемблер без труда найдет подходящую пару.

Часто эти две команды объединяются:

```
Z = TOS; if(Z) goto L1; else goto L2
```

В результате такой записи MAL сгенерирует микрокоманду, в которой значение регистра TOS пропускается через АЛУ, но при этом нигде не сохраняется, так что это значение устанавливает бит Z. Сразу после загрузки из АЛУ бит Z соединяется со старшим битом регистра MPC через схему ИЛИ, вследствие чего адрес следующей микрокоманды будет вызван либо по `L2`, либо по `L1`. Значение регистра MPC стабилизируется, и он сможет использовать его для вызова следующей микрокоманды.

Наконец, нам нужна специальная запись, чтобы задействовать бит JMPC:

```
goto (MBR OR значение)
```

Эта запись сообщает микроассемблеру, что нужно использовать *значение* для поля NEXT_ADDRESS и установить бит JMPC, так чтобы регистр MBR соединялся через схему ИЛИ с регистром MPC вместе со значением NEXT_ADDRESS. Если значение *value* равно 0, достаточно написать:

```
goto (MBR)
```

Отметим, что только 8 младших битов регистра MBR соединяются с регистром MPC (см. рис. 4.5), поэтому вопрос о знаковом расширении тут не возникает. Также отметим, что используется то значение MBR, которое доступно в конце текущего цикла. В момент вызова в *текущей* микрокоманде выбирать следующую микрокоманду уже поздно.

Реализация IJVM с использованием микроархитектуры Mic-1

Сейчас мы уже дошли до того момента, когда можно соединить все части вместе. В табл. 4.3 приводится микропрограмма, которая работает на микроархитектуре Mic-1 и интерпретирует IJVM. Программа очень короткая — всего 112 микрокоманд. Таблица состоит из трех столбцов. В первом столбце дано символическое обозначение микрокоманды, во втором — сама микрокоманда, в третьем — комментарий. Как мы уже отмечали, последовательность микрокоманд не обязательно соответствует последовательности адресов в управляющей памяти.

Таблица 4.3. Микропрограмма для микроархитектуры Mic-1

Микрокоманда	Операции	Комментарий
Main1	PC = PC + 1; fetch; goto(MBR)	MBR содержит код операции; получение следующего байта; отсылка
nop1	goto Main1	Ничего не происходит
iadd1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
iadd2	H = TOS	H = вершина стека
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Суммирование двух верхних слов; запись суммы в верхнюю позицию стека

Микро-команда	Операции	Комментарий
isub1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
isub2	H = TOS	H = вершина стека
isub3	MDR = TOS = MDR — H; wr; goto Main1	Вычитание; запись результата в вершину стека
iand1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
iand2	H = TOS	H = вершина стека
iand3	MDR = TOS = MDR И H; wr; goto Main1	Операция И; запись результата в вершину стека
ior1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
ior2	H = TOS	H = вершина стека
ior3	MDR = TOS = MDR ИЛИ H; wr; goto Main1	Операция ИЛИ; запись результата в вершину стека
dup1	MAR = SP = SP + 1	Увеличение SP на 1 и копирование результата в регистр MAR
dup2	MDR = TOS; wr; goto Main1	Запись нового слова в стек
pop1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
pop2		Программа ждет, пока считается из памяти новое значение регистра TOS
pop3	TOS = MDR; goto Main1	Копирование нового слова в регистр TOS
swap1	MAR = SP — 1; rd	Установка регистра MAR на значение SP — 1; чтение второго слова из стека
swap2	MAR = SP	Установка регистра MAR на верхнее слово стека
swap3	H = MDR; wr	Сохранение значения TOS в регистре H; запись второго слова в вершину стека
swap4	MDR = TOS	Копирование прежнего значения TOS в регистр MDR
swap5	MAR = SP — 1; wr	Установка регистра MAR на значение SP–1; запись второго слова в стек
swap6	TOS = H; goto Main1	Обновление TOS
bipush1	SP = MAR = SP + 1	MBR = байт, который нужно поместить в стек

продолжение ⇩

Таблица 4.3 *(продолжение)*

Микро-команда	Операции	Комментарий
bipush2	PC = PC + 1; fetch	Увеличение PC на 1; вызов кода следующей операции
bipush3	MDR = TOS = MBR; wr; goto Main1	Добавление к байту дополнительного знакового разряда и запись значения в стек
iload1	H = LV	MBR содержит индекс; копирование значения LV в H
iload2	MAR = MBRU + H; rd	MAR = адрес локальной переменной, которую нужно поместить в стек
iload3	MAR = SP = SP +1	Регистр SP указывает на новую вершину стека; подготовка к записи
iload4	PC = PC + 1; fetch; wr	Увеличение значения PC на 1; вызов кода следующей операции; запись вершины стека
iload5	TOS = MDR; goto Main1	Обновление TOS
istore1	H = LV	MBR содержит индекс; копирование значения LV в H
istore2	MAR = MBRU + H	MAR = адрес локальной переменной, в которой нужно сохранить слово из стека
istore3	MDR = TOS; wr	Копирование значения TOS в регистр MDR; запись слова
istore4	SP= MAR = SP — 1; rd	Чтение из стека второго слова сверху
istore5	PC = PC + 1; fetch	Увеличение PC на 1; вызов следующего кода операции
istore6	TOS = MDR; goto Main1	Обновление TOS
wide1	PC = PC + 1; fetch	Вызов байта операнда или кода следующей операции
wide2	goto(MBR ИЛИ 0x100)	Межуровневый переход к старшим адресам
wide_iload1	PC = PC + 1; fetch	MBR содержит первый байт индекса; вызов второго байта
wide_iload2	H = MBRU << 8	H = первый байт индекса, сдвинутый влево на 8 бит
wide_iload3	H=MBRU ИЛИ H	H = 16-разрядный индекс локальной переменной
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = адрес локальной переменной, которую нужно записать в стек
wide_istore1	PC = PC + 1; fetch	MBR содержит первый байт индекса; вызов второго байта

Микро-команда	Операции	Комментарий
wide_istore2	H = MBRU << 8	H = первый байт индекса, сдвинутый влево на 8 бит
wide_istore3	H = MBRU ИЛИ H	H = 16-разрядный индекс локальной переменной
wide_istore4	MAR = LV + H; goto istore3	MAR = адрес локальной переменной, в которую нужно записать слово из стека
ldc_w1	PC = PC + 1;fetch	MBR содержит первый байт индекса; вызов второго байта
ldc_w2	H = MBRU << 8	H = первый байт индекса, сдвинутый влево на 8 бит
ldc_w3	H = MBRU ИЛИ H	H = 16-разрядный индекс константы в наборе констант
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = адрес константы в наборе констант
iinc1	H = LV	MBR содержит индекс; копирование значения LV в H
iinc2	MAR = MBRU+H; rd	Копирование суммы значения LV и индекса в регистр MAR; чтение переменной
iinc3	PC = PC + 1; fetch	Вызов константы
iinc4	H = MDR	Копирование переменной в регистр H
iinc5	PC = PC + 1; fetch	Вызов следующего кода операции
iinc6	MDR = MBR + H; wr; goto Main1	Запись суммы в регистр MDR; обновление переменной
goto1	OPC = PC – 1	Сохранение адреса кода операции
goto2	PC = PC + 1; fetch	MBR = первый байт смещения; вызов второго байта
goto3	H = MBR << 8	Сдвиг первого байта влево на 8 бит и сохранение его в регистре H
goto4	H = MBRU ИЛИ H	H =16-разрядное смещение перехода
goto5	PC = OPC + H; fetch	Суммирование смещения и OPC
goto6	goto Main1	Ожидание вызова следующего кода операции
iflt1	MAR = SP = SP – 1; rd	Чтение второго сверху слова в стеке
iflt2	OPC = TOS	Временное сохранение TOS в OPC
iflt3	TOS = MDR	Запись новой вершины стека в TOS
iflt4	N = OPC; if(N) goto T; else goto F	Переход по биту N

продолжение ⇩

Таблица 4.3 *(продолжение)*

Микро-команда	Операции	Комментарий
ifeq1	MAR = SP = SP — 1; rd	Чтение второго сверху слова в стеке
ifeq2	OPC = TOS	Временное сохранение TOS в OPC
ifeq3	TOS = MDR	Запись новой вершины стека в TOS
ifeq4	Z = OPC; if(Z) goto T; else goto F	Переход по биту Z
if_icmpeq1	MAR = SP = SP — 1; rd	Чтение второго сверху слова в стеке
if_icmpeq2	MAR = SP = SP — 1	Установка регистра MAR на чтение новой вершины стека
if_icmpeq3	H = MDR; rd	Копирование второго слова из стека в регистр H
if_icmpeq4	OPC = TOS	Временное сохранение TOS в OPC
if_icmpeq5	TOS = MDR	Помещение новой вершины стека в TOS
if_icmpeq6	Z = OPC — H; if(Z) goto T; else goto F	Если два верхних слова равны, осуществляется переход к T; если они не равны, осуществляется переход к F
T	OPC = PC — 1; fetch; goto goto2	То же, что goto1; нужно для адреса целевого объекта
F	PC = PC + 1	Пропуск первого байта смещения
F2	PC = PC + 1; fetch	PC указывает на следующий код операции
F3	goto Main1	Ожидание вызова кода операции
invokevirtual1	PC = PC + 1; fetch	MBR = первый байт индекса; увеличение PC на 1; вызов второго байта
invokevirtual2	H = MBRU << 8	Сдвиг первого байта на 8 бит и сохранение значения в регистре H
invokevirtual3	H = MBRU ИЛИ H	H = смещение указателя процедуры от регистра CPP
invokevirtual4	MAR = CPP+H; rd	Вызов указателя процедуры из набора констант
invokevirtual5	OPC = PC + 1	Временное сохранение значения PC в регистре OPC
invokevirtual6	PC = MDR; fetch	Регистр PC указывает на новую процедуру; вызов числа параметров
invokevirtual7	PC = PC + 1; fetch	Вызов второго байта числа параметров
invokevirtual8	H = MBRU << 8	Сдвиг первого байта на 8 бит и сохранение значения в регистре H
invokevirtual9	H = MBRU ИЛИ H	H = число параметров

Микро-команда	Операции	Комментарий
invokevirtual10	PC = PC + 1; fetch	Вызов первого байта размера области локальных переменных
invokevirtual11	TOS = SP – H	TOS = адрес OBJREF–1
invokevirtual12	TOS = MAR = TOS + 1	TOS = адрес OBJREF (новое значение LV)
invokevirtual13	PC = PC + 1; fetch	Вызов второго байта размера области локальных переменных
invokevirtual14	H = MBRU << 8	Сдвиг первого байта на 8 бит и сохранение значения в регистре H
invokevirtual15	H = MBRU ИЛИ H	H = размер области локальных переменных
invokevirtual16	MDR = SP + H + 1; wr	Перезапись OBJREF со связующим указателем
invokevirtual17	MAR = SP = MDR	Установка регистров SP и MAR на адрес ячейки, в которой содержится старое значение PC
invokevirtual18	MDR = OPC; wr	Сохранение старого значения PC над локальными переменными
invokevirtual19	MAR = SP = SP + 1	SP указывает на ячейку, в которой хранится старое значение LV
invokevirtual20	MDR = LV; wr	Сохранение старого значения LV над сохраненным значением PC
invokevirtual21	PC = PC + 1; fetch	Вызов первого кода операции новой процедуры
invokevirtual22	LV = TOS; goto Main1	Установка значения LV на первый адрес кадра локальных переменных
ireturn1	MAR = SP = LV; rd	Сброс регистров SP и MAR для получения связующего указателя
ireturn2		Ожидание считывания
ireturn3	LV = MAR = MDR; rd	Установка регистра LV на связующий указатель; вызов старого значения PC
ireturn4	MAR = LV + 1	Установка регистра MAR на чтение старого значения LV
ireturn5	PC = MDR; rd; fetch	Восстановление PC; вызов следующего кода операции
ireturn6	MAR = SP	Установка MAR на запись TOS
ireturn7	LV = MDR	Восстановление LV
ireturn8	MDR = TOS; wr; goto Main1	Сохранение результата в исходной вершине стека

Выбор названий для большинства регистров, изображенных на рис. 4.1, должен быть очевидным. Регистры CPP (Constant Pool Pointer — указатель набора констант), LV (Local Variable — локальная переменная) и SP (Stack Pointer — указатель стека) содержат указатели на адреса набора констант, кадра локальных переменных и верхнего элемента в стеке соответственно, а регистр PC (Program Counter — счетчик команд) содержит адрес байта, который нужно вызвать из потока команд следующим. Регистр MBR (Memory Buffer Register — буферный регистр памяти) — это 1-байтный регистр, который содержит байты потока команд, поступающих из памяти для интерпретации. TOS и OPC — дополнительные регистры. Они описываются далее.

В определенные моменты в каждом из этих регистров обязательно находится определенное значение. Однако каждый из них в случае необходимости может также использоваться в качестве временного. В начале и конце каждой команды регистр TOS (Top Of Stack — вершина стека) содержит значение адреса памяти, на который указывает SP. Это значение избыточно, поскольку его всегда можно считать из памяти, но если хранить это значение в регистре, то обращение к памяти не потребуется. Для некоторых команд использование регистра TOS, напротив, влечет за собой *увеличение* количества обращений к памяти. Например, команда POP отбрасывает верхнее слово стека и, следовательно, должна вызвать новое значение вершины стека из памяти и записать его в регистр TOS.

OPC — временный регистр. У него нет определенного назначения. В нем, например, может храниться адрес кода операции для команды перехода, пока значение PC увеличивается, чтобы получить доступ к параметрам. Он также используется в качестве временного регистра в командах условного перехода.

Как и все интерпретаторы, микропрограмма, приведенная в табл. 4.3, включает в себя основной цикл, который вызывает, декодирует и выполняет команды интерпретируемой программы (в данном случае IJVM-команды). Основной цикл начинается со строки `Main1`, а именно — с инварианта (обязательного условия), что в регистр PC уже загружен адрес ячейки памяти, в которой содержится код операции. Более того, этот код операции уже вызван из памяти в регистр MBR. Когда мы вернемся к этой ячейке, мы должны быть уверены, что значение PC уже обновлено и указывает на код следующей операции, а сам код операции уже вызван из памяти в MBR.

Такая последовательность действий имеет место в начале каждой команды, поэтому важно сделать ее как можно более короткой. Разрабатывая аппаратное и программное обеспечение микроархитектуры Mic-1, мы смогли сократить основной цикл до одной микрокоманды. Каждый раз, когда выполняется эта микрокоманда, код выполняемой операции уже находится в регистре MBR. Эта микрокоманда, во-первых, осуществляет переход к микрокоду, выполняющему данную операцию, во-вторых, вызывает следующий после кода операции байт, который может быть либо операндом, либо кодом операции.

Теперь мы можем открыть главную причину того, почему в каждой микрокоманде явным образом указывается следующая микрокоманда, а последовательность команд может не соответствовать порядку их расположения в памяти. Все адреса управляющей памяти, соответствующие кодам операций, должны быть зарезервированы для первого слова интерпретатора соответствующей команды.

Так, из табл. 4.2 мы видим, что программа, которая интерпретирует команду POP, начинается в ячейке 0x57, а программа, которая интерпретирует команду DUP, начинается в ячейке 0x59. (Как язык MAL узнает, что команду POP нужно поместить в ячейку 0x57? Наверное, из какого-нибудь файла.)

К сожалению, программа, интерпретирующая команду POP, включает в себя три микрокоманды, поэтому если их расположить в памяти последовательно, то эта программа смешается с началом команды DUP. Поскольку все адреса управляющей памяти, которые соответствуют кодам операций, зарезервированы, то все микрокоманды, идущие после первой микрокоманды в каждой последовательности, должны размещаться в промежутках между зарезервированными адресами. По этой причине происходит очень много «скачков», и было бы нерационально каждый раз вставлять микрокоманду перехода, чтобы перейти от одной последовательности адресов к другой.

Чтобы понять, как работает интерпретатор, предположим, что регистр MBR содержит значение 0x60, то есть код операции IADD (см. табл. 4.2). В основном цикле, который состоит из одной микрокоманды, выполняется следующее:

1. Значение регистра PC увеличивается, после чего он содержит адрес первого байта после кода операции.
2. Начинается передача следующего байта в регистр MBR. Этот байт понадобится рано или поздно либо в качестве операнда текущей IJVM-команды, либо в качестве кода следующей операции (как в случае с командой IADD, у которой нет операндов).
3. Совершается переход к адресу, который содержался в регистре MBR в начале цикла Main1. Номер адреса равен значению кода операции, которая выполняется в данный момент. Этот адрес помещается туда предыдущей микрокомандой. Отметим, что значение, которое вызывается из памяти во время этой микрокоманды, не играет никакой роли в межуровневом переходе.

Здесь начинается вызов следующего байта, поэтому он будет доступен уже к концу третьей микрокоманды. Хотя в тот момент указанный байт может не нужен, в любом случае когда-нибудь он понадобится, поэтому не будет никакого вреда в том, что вызов происходит именно здесь.

Если все разряды байта в регистре MBR равны 0 (это код операции для команды NOP), то следующей будет микрокоманда nop1, которая вызывается из ячейки 0. Поскольку эта команда не производит никаких операций, она просто совершает переход к началу основного цикла, где повторяется та же последовательность действий, но уже с новым кодом операции в MBR.

Еще раз подчеркнем, что микрокоманды, приведенные в табл. 4.3, расположены в памяти не последовательно, а микрокоманда Main1 вовсе не находится в ячейке с адресом 0 (поскольку в этой ячейке должна находиться микрокоманда nop1). Задача микроассемблера — поместить каждую команду в подходящую ячейку и связать их в короткие последовательности, используя поле NEXT_ADDRESS. Каждая последовательность начинается с адреса, который соответствует численному значению кода операции (например, команда POP начинается с адреса 0x57), но остальные части последовательности могут находиться в любых ячейках управляющей памяти, и эти ячейки не обязательно смежные.

А теперь рассмотрим команду IADD. Она начинается с микрокоманды iadd1. Требуется выполнить следующие действия:

1. Значение регистра TOS уже есть, но из памяти нужно вызвать второе слово стека.
2. Значение регистра TOS нужно прибавить ко второму слову стека, вызванному из памяти.
3. Результат, который помещается в стек, должен быть сохранен в памяти и в регистре TOS.

Чтобы вызвать операнд из памяти, необходимо уменьшить значение указателя стека и записать его в регистр MAR. Отметим, что этот адрес будет использоваться для последующей записи. Более того, поскольку эта ячейка памяти станет новой вершиной стека, данное значение должно быть присвоено регистру SP. Следовательно, определить новое значение SP и MAR, уменьшить значение SP на 1 и записать его в оба регистра можно за одну операцию.

Все эти действия выполняются в первом цикле (iadd1). Здесь же инициируется операция чтения. Кроме того, регистр MPC получает значение из поля NEXT_ADDRESS микрокоманды iadd1. Это адрес микрокоманды iadd2. Затем iadd2 считывается из управляющей памяти. Во втором цикле, пока происходит считывание операнда из памяти, мы копируем верхнее слово стека из TOS в H, где оно будет доступно для сложения, когда процесс считывания завершится.

В начале третьего цикла (iadd3) MDR содержит второе слагаемое, вызванное из памяти. В этом цикле оно прибавляется к значению регистра H, а результат сохраняется обратно в регистрах MDR и TOS. Кроме того, начинается операция записи, в процессе которой новое верхнее слово стека сохраняется в памяти. В этом цикле команда goto приписывает адрес Main1 регистру MPC, и таким образом мы возвращаемся к исходному пункту и можем начать выполнение следующей операции.

Если следующий код IJVM-операции, который содержится в данный момент в регистре MBR, равен 0x64 (ISUB), то повторяется практически та же последовательность действий. После выполнения Main1 управление передается микрокоманде с адресом 0x64 (isub1). За этой микрокомандой следуют isub2, isub3, а затем снова Main1. Единственное отличие между этой и предыдущей последовательностями состоит в том, что в цикле isub3 содержимое регистра H не прибавляется к значению MDR, а вычитается из него.

Команда IAND идентична командам IADD и ISUB, только в данном случае два верхних слова стека подвергаются логическому умножению (операция И), а не складываются и не вычитаются. Нечто подобное происходит и во время выполнения команды IOR.

Если код операции соответствует команде DUP, POP или SWAP, то нужно использовать стек. Команда DUP дублирует верхнее слово стека. Поскольку значение этого слова уже находится в регистре TOS, нужно просто увеличить SP на 1. После этого регистр SP будет указывать на новый адрес. В эту новую ячейку и записывается значение регистра TOS. Команда POP тоже достаточно проста: нужно только уменьшить значение SP на 1, чтобы отбросить верхнее слово стека. Однако после этого необходимо считать новое верхнее слово стека из памяти и записать его в регистр TOS. Наконец, команда SWAP меняет местами значения

двух ячеек памяти, а именно — два верхних слова стека. Регистр TOS уже содержит одно из этих значений, поэтому считывать его (значение) из памяти не нужно. Подробнее мы обсудим эту команду немного позже.

Команда `BIPUSH` сложнее предыдущих, поскольку за кодом операции следует байт, как показано на рис. 4.13. Этот байт интерпретируется как целое число со знаком. Указанный байт, который был передан в регистр MBR во время выполнения микрокоманды `Main1`, нужно расширить до 32 бит (знаковое расширение) и скопировать его в регистр MDR. Затем значение SP увеличивается на 1 и копируется в MAR, что позволяет записать операнд на вершину стека. Этот операнд также должен копироваться в регистр TOS. Отметим, что перед возвращением управления в основную программу значение регистра PC должно быть увеличено на 1 и начата операция выборки, чтобы в `Main1` был доступен код следующей операции.

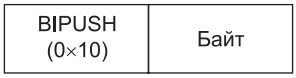


Рис. 4.13. Формат команды BIPUSH

Теперь рассмотрим команду `ILOAD`. В этой команде за кодом операции также следует байт (рис. 4.14, *а*), но этот байт представляет собой индекс (без знака), используемый для того, чтобы найти в пространстве локальных переменных слово, которое нужно поместить в стек. Поскольку здесь имеется всего 1 байт, доступно только $2^8 = 256$ слов, а именно первые 256 слов пространства локальных переменных. Для выполнения команды `ILOAD` требуются и чтение (чтобы вызвать слово), и запись (чтобы поместить его в стек). Чтобы определить адрес для считывания, нужно прибавить смещение, которое хранится в регистре MBR (это буферный регистр памяти), к содержимому регистра LV. Доступ к регистрам MBR и LV можно получить только через шину B, поэтому сначала значение LV копируется в регистр H (в цикле `iload1`), а затем прибавляется значение MBR. Результат суммирования копируется в регистр MAR, и начинается процесс чтения (в цикле `iload2`).

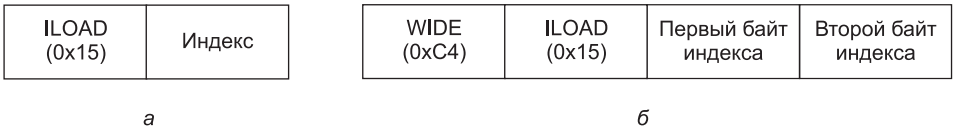


Рис. 4.14. Команда ILOAD с однобайтным индексом (*а*); команда WIDE ILOAD с двухбайтным индексом (*б*)

Однако здесь регистр MBR используется не совсем так, как в команде `BIPUSH`, где байт расширялся со знаком. В случае с индексом смещение всегда положительно, поэтому смещение в байтах должно быть целым числом без знака (в отличие от `BIPUSH`, где байт интерпретируется как 8-разрядное целое число со знаком). Интерфейс между регистром MBR и шиной B разработан таким образом, чтобы были возможны обе операции. В случае с командой `BIPUSH` (где байт — 8-разрядное целое число со знаком) самый левый бит значения MBR копируется в 24 старших бита шины B. В случае с командой `ILOAD` (где байт — 8-разрядное целое число без знака) 24 старших бита шины B заполняются

нулями. Два специальных сигнала помогают определить, какую из этих двух операций нужно выполнить (см. рис. 4.5). В микропрограмме слово MBR указывает на байт со знаком (как в команде `bipush3`), а MBRU — на байт без знака (как в команде `iload2`).

Пока ожидается поступление операнда из памяти (во время `iload3`), значение регистра SP увеличивается на 1 для записи новой вершины стека. Это значение также копируется в регистр MAR (это требуется для записи операнда в стек). Затем значение PC снова увеличивается на 1 для выборки следующего кода операции (микрокоманда `iload4`). Наконец, значение MDR копируется в регистр TOS, чтобы показать новое верхнее слово стека (микрокоманда `iload5`).

Команда `ISTORE` противоположна команде `ILOAD` (из стека выталкивается верхнее слово и сохраняется в ячейке памяти, адрес которой равен сумме значения регистра LV и индекса данной команды). В данном случае используется такой же формат, как и в команде `ILOAD` (см. рис. 4.14, *а*), только здесь код операции равен не 0x15, а 0x36. Поскольку верхнее слово стека уже известно (оно находится в регистре TOS), его можно сразу сохранить в памяти. Однако новое верхнее слово стека все же необходимо вызвать из памяти, поэтому требуются и чтение, и запись, хотя эти операции можно выполнять в любом порядке (или даже одновременно, если бы это было возможно).

Команды `ILOAD` и `ISTORE` имеют доступ только к первым 256 локальным переменным. Хотя для большинства программ этого пространства достаточно, все же нужно иметь возможность обращаться к любой локальной переменной, в какой бы части кадра она ни находилась. Чтобы обеспечить такую возможность, IJVM использует то же средство, что и JVM — специальный код операции `WIDE` (так называемый **префиксный байт**), за которым следует код операции `ILOAD` или `ISTORE`. Когда встречается такая последовательность, формат команды `ILOAD` или `ISTORE` меняется; в соответствие с новым форматом за кодом операции следует не 8-разрядный, а 16-разрядный индекс, как показано на рис. 4.14, *б*.

Команда `WIDE` декодируется обычным способом. Сначала происходит переход к микрокоманде `wide1`, которая обрабатывает код операции команды `WIDE`. Хотя код операции, который нужно расширить, уже присутствует в регистре MBR, микрокоманда `wide1` вызывает первый байт после кода операции, поскольку этого требует логика микропрограммы. Затем совершается еще один межуровневый переход, но на этот раз для перехода используется байт, следующий за `WIDE`. Однако поскольку команда `WIDE ILOAD` требует иного чем `ILOAD` набора микрокоманд, команда `WIDE ISTORE` — иного чем `ISTORE` и т. д., при втором межуровневом переходе код операции нельзя использовать в качестве целевого адреса, как это делается в `Main1`.

Вместо этого микрокоманда `wide1` подвергает логическому сложению адрес 0x100 и код операции, поместив его в регистр MPC. В результате интерпретация `WIDE LOAD` начинается с адреса 0x115 (а не 0x15), интерпретация `WIDE ISTORE` — с адреса 0x136 (а не 0x36) и т. д. Таким образом, каждый код операции `WIDE` начинается с адреса, который в управляющей памяти на 256 (то есть 0x100) слов выше, чем соответствующий код обычной операции. Начальная последовательность микрокоманд для `ILOAD` и `WIDE ILOAD` показана на рис. 4.15.

Команда `WIDE ILOAD` отличается от обычной команды `ILOAD` только тем, что индекс в ней состоит из двух индексных байтов. Слияние и последующее сум-

Рис. 4.15. Начало последовательности микрокоманд для команд ILOAD и WIDE ILOAD. Адреса приводятся в качестве примера

мирование этих байтов должно происходить поэтапно, при этом сначала первый индексный байт сдвигается влево на 8 бит и копируется в H. Поскольку индекс — целое число без знака, здесь используется регистр MBRU (24 старших бита заполняются нулями). Затем прибавляется второй байт индекса (операция сложения идентична слиянию, поскольку младший байт регистра H в данный момент равен 0), при этом гарантируется, что между байтами не будет переноса. Результат снова сохраняется в регистре H. С этого момента происходят те же действия, что и в стандартной команде `ILOAD`. Вместо того чтобы дублировать последние микрокоманды (от `iload3` до `iload5`) команды `ILOAD`, мы просто совершили переход от `wide_iload4` к `iload3`. Отметим, что во время выполнения этой команды значение PC должно увеличиваться на 1 дважды, чтобы в конце этот регистр указывал на следующий код операции. Команда `ILOAD` увеличивает значение один раз; последовательность команд `WIDE_ILOAD` также увеличивает это значение один раз.

Такая же ситуация имеет место при выполнении `WIDE_ISTORE`. После первых четырех микрокоманд (от `wide_istore1` до `wide_istore4`) последовательность действий та же, что и в команде `ISTORE` после первых двух микрокоманд, поэтому мы совершаем переход от `wide_istore4` к `istore3`.

Далее мы рассмотрим команду `LDC_W`. Между этой командой и `ILOAD` существует два отличия. Во-первых, она содержит 16-разрядное смещение без знака (как и расширенная версия `ILOAD`), во-вторых, эта команда индексируется из регистра CPP, а не из LV, поскольку она считывает значение из набора констант, а не из кадра локальных переменных. (Существует еще и краткая форма этой команды — `LDC`, но мы не стали включать ее в машину IJVM, поскольку полная

форма содержит в себе все варианты краткой формы, хотя при этом занимает 3 байта вместо 2.)

Команда `IINC` — единственная команда, помимо `ISTORE`, которая может изменять локальную переменную. Она включает в себя два операнда по одному байту, как показано на рис. 4.16.

Рис. 4.16. Команда IINC содержит два поля операндов

Поле индекса нужно для того, чтобы определить смещение от начала кадра локальных переменных. Команда считывает эту переменную, увеличивает ее на константу (константа содержится во втором поле) и помещает результат обратно в ту же ячейку памяти. Отметим, что константа является 8-разрядным числом со знаком в промежутке от –128 до +127. Машина JVM поддерживает расширенную версию этой команды, в которой длина каждого операнда составляет 2 байта.

Рассмотрим первую команду перехода — `GOTO`. Эта команда изменяет значение регистра PC таким образом, чтобы следующая IJVM-команда находилась в ячейке памяти с адресом, который вычисляется путем прибавления 16-разрядного смещения (со знаком) к адресу кода операции `GOTO`. Сложность здесь в том, что смещение связано со значением, находящимся в регистре PC в начале декодирования команды, а не тем, которое содержится в том же регистре после вызова 2 байт смещения.

Чтобы лучше это понять, посмотрите на рис. 4.17, *а*. Здесь показана ситуация, которая имеет место в начале цикла `Main1`. Код операции уже находится в регистре MBR, но значение PC еще не увеличилось. На рис. 4.17, *б* мы видим ситуацию в начале цикла `goto1`. В данном случае значение PC уже увеличено на 1, а первый байт смещения уже передан в MBR. В следующей микрокоманде (рис. 4.17, *в*) прежнее значение PC, которое указывает на код операции, сохраняется в регистре OPC. Это значение требуется сохранять, поскольку именно от него, а не от текущего значения PC зависит смещение команды `GOTO`. И именно для этого предназначен регистр OPC.

Микрокоманда `goto2` начинает вызов второго байта смещения, что приводит к ситуации, показанной на рис. 4.17, *г* (микрокоманда `goto3`). После того как первый байт смещения сдвигается влево на 8 бит и копируется в регистр H, мы переходим к микрокоманде `goto4` (рис. 4.17, *д*). Теперь у нас первый байт смещения, сдвинутый влево, находится в регистре H, второй байт смещения — в регистре MBR, а основание смещения — в регистре OPC. В микрокоманде `goto5` путем прибавления полного 16-разрядного смещения к основанию смещения мы получаем новый адрес, который помещается в регистр PC. Отметим, что в `goto4` вместо MBR используется регистр MBRU, поскольку нам не требуется знаковое расширение второго байта. 16-разрядное смещение строится путем логического сложения (операция ИЛИ) двух половинок. Наконец, поскольку программа перед переходом к `Main1` требует, чтобы в MBR был помещен код следующей операции, мы должны вызвать этот код. Последний цикл, `goto6`, нужен для того, чтобы вовремя помесить данные из памяти в регистр MBR.

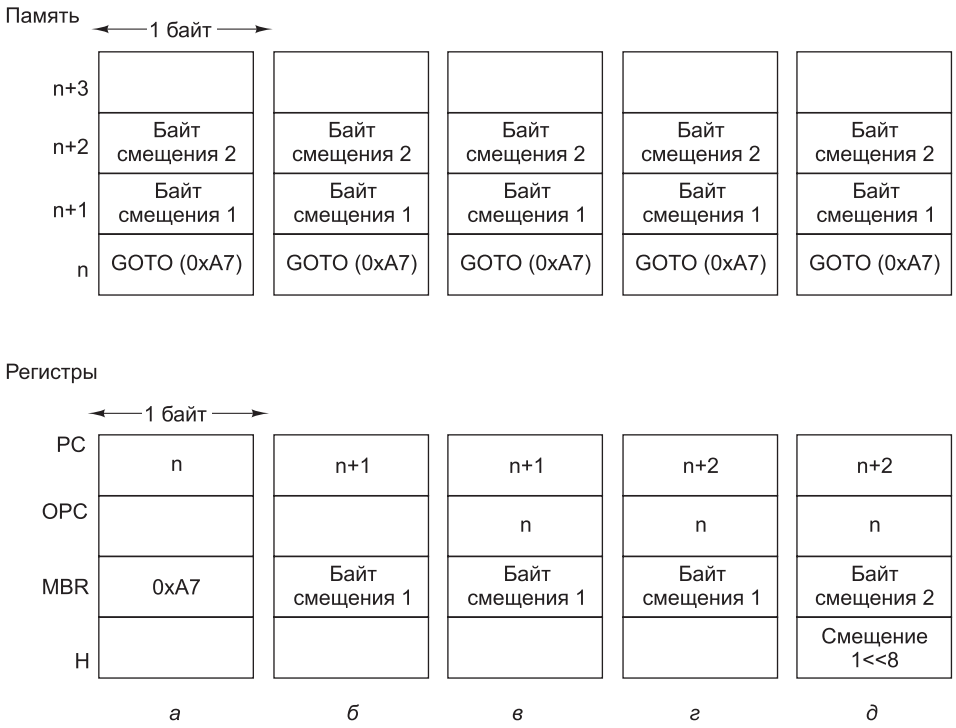


Рис. 4.17. Ситуация в начале выполнения различных микрокоманд: Main1 (*а*); goto1 (*б*); goto2 (*в*); goto3 (*г*); goto4 (*д*)

Смещения, которые используются в команде `goto`, представляют собой 16-разрядные значения со знаком в промежутке от –32 768 до +32 767. Это значит, что переходы на более дальние расстояния невозможны. Это свойство можно рассматривать либо как дефект, либо как особенность машины IJVM (а также JVM). Те, кто считает это дефектом, скажут, что машина JVM не должна ограничивать стиль программирования. Те, кто считает это особенностью, скажут, что работа многих программистов продвинулась бы кардинальным образом, если бы им в ночных кошмарах приснилось следующее сообщение компилятора:

Программа слишком длинная и сложная. Вы должны переписать ее. Компиляция прекращена.

К сожалению (как мы считаем), это сообщение появится только в том случае, если объем предложения `else` или `then` превысит 32 Кбайт, что составляет по крайней мере 50 страниц текста на языке Java.

А теперь рассмотрим три команды условного перехода: `IFLT`, `IFEQ` и `IF_ICMPEQ`. Первые две выталкивают верхнее слово из стека и совершают переход в том случае, если это слово меньше 0 или равно 0 соответственно. Команда `IF_ICMPEQ` берет два верхних слова из стека и совершает переход, если они равны. Во всех трех случаях необходимо считывать новое верхнее слово стека и помещать его в регистр TOS.

Эти три команды сходны. Сначала операнд или операнды помещаются в регистры, затем в TOS записывается новое верхнее слово стека, и наконец, происходит сравнение и осуществляется переход. Сначала рассмотрим `IFLT`. Слово, которое нужно проверить, уже находится в регистре TOS, но поскольку команда `IFLT` выталкивает слово из стека, нужно считать из памяти новую вершину стека и сохранить ее в регистре TOS. Процесс считывания начинается в микрокоманде `iflt1`. Во время `iflt2` проверяемое слово сохраняется в регистре OPC, поэтому новое значение можно сразу помесить в регистр TOS, и при этом предыдущее значение не пропадет. В цикле `iflt3` новое верхнее слово стека, которое уже находится в MDR, копируется в регистр TOS. Наконец, в цикле `iflt4` проверяемое слово (оно находится в регистре OPC) пропускается через АЛУ без сохранения результата, после чего проверяется бит N. Если после проверки условие подтверждается, микрокоманда осуществляет переход к `T`, а если не подтверждается — к `F`.

Если условие подтверждается, то происходят те же действия, что и в начале команды `GOTO`, и далее осуществляется переход к `goto2`. Если условие не подтверждается, необходима короткая последовательность микрокоманд (`F`, `F2` и `F3`), чтобы пропустить оставшуюся часть команды (смещение), возвратиться к `Main1` и перейти к следующей команде.

Команды `ifeq2` и `ifeq3` выглядят аналогично, только вместо бита N используется бит Z. В обоих случаях ассемблер должен убедиться, что адреса `F` и `T` размещаются в адресах управляющей памяти, отличающихся только крайним левым битом.

Команда `IF_ICMPEQ` в целом похожа на команду `IFLT`, только здесь нужно считывать еще и второй операнд. Второй операнд сохраняется в регистре H во время цикла `if_icmpeq3`, где начинается чтение нового верхнего слова стека. Текущее верхнее слово стека сохраняется в OPC, а новое загружается в регистр TOS. Наконец, микрокоманда `if_icmpeq6` аналогична `ifeq4`.

Теперь рассмотрим команды `INVOKEVIRTUAL` и `IRETURN`. Как отмечалось в подразделе «Набор IJVM-команд» раздела «Пример архитектуры набора команд — IJVM», они служат для вызова процедуры и выхода из нее. Команда `INVOKEVIRTUAL` представляет собой последовательность из 22 микрокоманд — самая сложная реализация команды IJVM. Последовательность действий при выполнении этой команды иллюстрирует рис. 4.10. 16-разрядное смещение используется для определения адреса вызываемой процедуры. Соответствующий элемент набора констант указывает на вызываемую процедуру. Следует помнить, что первые 4 байта каждой процедуры — *не* команды, а два 16-разрядных указателя. Первый из них позволяет узнать число параметров (включая OBJREF — см. рис. 4.10), второй — размер области локальных переменных (в словах). Эти поля вызываются через 8-разрядный порт и объединяются таким же образом, как 16-разрядное смещение в одной команде.

Затем специальная информация, необходимая для восстановления предыдущего состояния машины, — адрес начала прежней области локальных переменных и старое значение регистра PC — сохраняется непосредственно над областью локальных переменных под новым стеком. Наконец, вызывается следующий код операции, значение регистра PC увеличивается, происходит переход к циклу `Main1` и начинается выполнение следующей команды.

`IRETURN` — простая команда без операндов. Эта команда просто использует адрес, хранящийся в первом слове области локальных переменных, чтобы извлечь

информацию для возвращения к прежнему состоянию. Затем она восстанавливает предыдущие значения регистров SP, LV и PC и копирует результат выполнения процедуры из нового стека в предыдущий стек, как показано на рис. 4.11.

Разработка уровня микроархитектуры

При разработке уровня микроархитектуры (как и при разработке других уровней) постоянно приходится идти на компромисс. У компьютера есть много важных характеристик: быстродействие, стоимость, надежность, простота использования, объем потребляемой энергии, физические размеры. При разработке центрального процессора очень важную роль играет правильный выбор между быстродействием и стоимостью. В этом разделе мы подробно рассмотрим данную дилемму, покажем преимущества и недостатки каждого из вариантов, а также узнаем, какой производительности можно достичь, какова при этом будет стоимость компьютера и насколько сложным окажется аппаратное обеспечение.

Быстродействие и стоимость

С развитием технологий быстродействие компьютеров стремительно растет. В основном этот процесс проходит благодаря увеличению скорости работы микросхем, хотя архитектурный фактор также оказывает на него определенное влияние. Существуют три основных подхода, которые позволяют увеличить скорость выполнения операций:

1. Сократить количество циклов, необходимых для выполнения команды.
2. Упростить организацию машины таким образом, чтобы можно было сделать цикл короче.
3. Сделать так, чтобы несколько операций выполнялось одновременно.

Первые два подхода очевидны, но существует огромное количество различных вариантов разработки, которые могут очень значительно повлиять на число циклов, их продолжительность или (что бывает чаще всего) на то и другое вместе. В этом разделе мы приведем пример того, как кодирование и декодирование операции могут действовать на цикл.

Число циклов, необходимых для выполнения набора операций, называется **длиной пути**. Иногда длину пути можно уменьшить с помощью дополнительного аппаратного обеспечения. Например, если к регистру PC добавить схему инкремента (по сути, сумматор, у которого один из входов постоянно связан с единицей), то нам больше не придется использовать для увеличения значения PC на единицу АЛУ, и, следовательно, количество циклов сократится. Однако такой подход не настолько эффективен, как хотелось бы. Часто в том же цикле, в котором значение PC увеличивается на единицу, происходит еще и операция чтения, и следующая команда в любом случае не может начаться раньше, поскольку она зависит от данных, которые должны поступить из памяти.

Для сокращения числа циклов, необходимых для вызова команды, требуется нечто большее, чем простое добавление схемы, которая увеличивает PC на 1. Чтобы повысить скорость вызова команды, нужно применить третий подход — параллельное выполнение команд. Весьма эффективно отделение схем

для вызова команд (8-разрядного порта памяти и регистров PC и MBR), если этот блок сделать функционально независимым от основного тракта данных. Таким образом, он может сам вызывать следующий код операции или операнд. Возможно, он даже будет работать асинхронно относительно другой части процессора и осуществлять выборку одной или нескольких команд заранее.

Один из наиболее трудоемких процессов при выполнении команд — вызов 2-байтного смещения, его соответствующее расширение и сохранение в регистре H для подготовки к сложению (например, при переходе к PC ± *n* байт). Одно из возможных решений — увеличить порт памяти до 16 бит, но это значительно усложняет операцию, поскольку требуемые 16 бит могут выходить за границы слова, тогда даже считывание из памяти 32 бит за одно обращение не обязательно приведет к вызову обоих нужных нам байтов.

Одновременное выполнение нескольких операций — самый продуктивный подход. Он дает возможность значительно увеличить быстродействие компьютера. Даже простое перекрытие вызова и выполнения команды чрезвычайно эффективны. При более сложных технологиях допустимо одновременное выполнение нескольких команд. Вообще говоря, эта идея является основой проектов современных компьютеров. Далее мы обсудим некоторые технические приемы, позволяющие воплотить этот подход в жизнь.

На одной чаше весов находится быстродействие, на другой — стоимость. Стоимость можно измерять различными способами, но точное определение стоимости дать очень трудно. В те времена, когда процессоры собирались из дискретных компонентов, достаточно было подсчитать общее число этих компонентов. В настоящее время процессор целиком помещается на одну микросхему, но большие и более сложные микросхемы стоят гораздо дороже, чем более простые микросхемы небольшого размера. Можно посчитать отдельные компоненты (транзисторы, вентили, функциональные блоки), но обычно это число не так важно, как размер контактного участка, необходимый для интегральной схемы. Чем больше участок, тем больше микросхема. И стоимость микросхемы растет гораздо быстрее, чем занимаемое ею пространство. По этой причине разработчики часто измеряют стоимость в единицах, применимых к «недвижимости», то есть с точки зрения пространства, которое требуется для микросхемы (предполагаем, что площадь поверхности измеряется в пикоакрах).

В истории компьютерной индустрии одной из наиболее тщательно проработанных микросхем является двоичный сумматор. Были реализованы тысячи разных конструкций, и самые быстрые двоичные сумматоры значительно превосходят по быстродействию наиболее медленные. Естественно, высокоскоростные сумматоры гораздо сложнее низкоскоростных. Специалистам по разработке систем приходится останавливаться на определенном соотношении быстродействия и занимаемого пространства.

Сумматор — не единственный компонент, допускающий различные варианты разработки. Практически любой компонент системы может быть спроектирован таким образом, что он будет функционировать с более высокой или с более низкой скоростью, при этом, естественно, разные модели будут различаться по стоимости. Главной задачей разработчика является определение тех компонентов системы, усовершенствование которых может максимально повлиять на быстродействие компьютера. Интересно отметить, что если какой-нибудь компонент заменить более быстрым, это не обязательно повлечет за собой рост общей про-

изводительности. В следующих подразделах мы рассмотрим некоторые аспекты разработки и возможные соотношения стоимости и быстродействия.

Одним из ключевых факторов в определении скорости работы генератора синхронизирующего сигнала является количество действий, которые должны быть сделаны за один цикл. Очевидно, чем больше действий должно быть сделано, тем длиннее цикл. Однако все не так просто, ведь аппаратное обеспечение способно выполнять некоторые операции параллельно, поэтому в действительности длина цикла зависит от количества *последовательных* операций в одном цикле.

Также необходимо учитывать объем декодирования. Посмотрите на рис. 4.5. Вспомните, что в АЛУ может передаваться значение одного из девяти регистров, и чтобы определить, какой именно регистр нужно выбрать, требуется всего 4 бита в микрокоманде. К сожалению, такая экономия дорого обходится. Схема декодера вносит дополнительную задержку в работу компьютера. Это значит, что какой бы регистр мы ни выбрали, он получит команду (и передаст свое содержимое на шину B) немного позже. Следовательно, АЛУ получает входные сигналы и выдает результат также с небольшой задержкой. Соответственно, этот результат тоже поступает на шину C для записи в один из регистров чуть позже. Поскольку задержка часто является фактором, который определяет длину цикла, это значит, что генератор синхронизирующего сигнала не сможет функционировать с такой скоростью, и весь компьютер должен работать немного медленнее. Таким образом, существует определенная зависимость между быстродействием и стоимостью. Если сократить каждое слово управляющей памяти на 5 бит, это приведет к снижению скорости работы генератора. Инженер при разработке компьютера должен принимать во внимание его предназначение, чтобы сделать правильный выбор. В компьютере с высокой производительностью использовать декодер не рекомендуется, а вот для дешевой машины он вполне подойдет.

Сокращение длины пути

Микроархитектура Mic-1 имеет относительно простую структуру и работает довольно быстро, хотя эти две характеристики очень сильно конфликтуют друг с другом. Проще говоря, простые машины не так быстры, а быстрые — не так просты, как нам хотелось бы. В процессоре Mic-1 используется минимум аппаратного обеспечения: 10 регистров, простое АЛУ (см. рис. 3.18), продублированное 32 раза, декодер, схема сдвига, управляющая память и некоторые связующие элементы. Для построения всей системы требуется менее 5000 транзисторов, плюс управляющая память (ПЗУ), плюс основная память (ОЗУ).

Мы уже показали, как можно воплотить IJVM микропрограммно, используя минимум аппаратуры. Теперь рассмотрим альтернативные варианты. Сначала мы выясним, какими способами можно снизить количество микрокоманд в одной команде (то есть каким образом можно сократить длину пути), а затем перейдем к другим подходам.

Слияние цикла интерпретатора с микропрограммой

В микроархитектуре Mic-1 основной цикл состоит из микрокоманды, которая должна выполняться в начале каждой IJVM-команды. В некоторых случаях допустимо ее перекрытие предыдущей командой. В каком-то смысле эта идея уже получила свое воплощение. Вспомните, что во время цикла `Main1` код следующей

операции уже находится в регистре MBR. Этот код операции вызывается либо во время предыдущего основного цикла (если у предыдущей команды не было операндов), либо при выполнении предыдущей команды.

Концепцию перекрытия начала команды можно развивать и дальше. В некоторых случаях основной цикл можно свести к нулю. Это происходит следующим образом. Рассмотрим каждую последовательность микрокоманд, которая завершается переходом к `Main1`. Каждый раз основной цикл может добавляться в конце этой последовательности (а не в начале следующей), при этом межуровневый переход дублируется много раз (но всегда с одним и тем же набором целевых объектов). В некоторых случаях микрокоманда микроархитектуры Mic-1 может сливаться с предыдущими микрокомандами, поскольку эти команды используются не всегда полностью.

В табл. 4.4 приведена последовательность выполнения микрокоманд для команды `POP`. Основной цикл идет перед каждой командой и после каждой команды, в таблице этот цикл показан только после команды `POP`. Отметим, что выполнение этой команды занимает 4 цикла: три цикла специальных микрокоманд команды `POP` и один основной цикл.

Таблица 4.4. Новая микропрограмма для выполнения команды POP

Микрокоманда	Операции	Комментарий
pop1	MAR = SP = SP — 1; rd	Считывание второго сверху слова в стеке
pop2		Ожидание, пока из памяти считается новое значение TOS
pop3	TOS = MDR; goto Main1	Копирование нового слова в регистр TOS
Main1	PC = PC + 1; fetch; goto(MBR)	Регистр MBR содержит код операции; вызов следующего байта; переход

В табл. 4.5 последовательность сокращена до трех команд за счет того, что в цикле `pop2` АЛУ не используется. Отметим, что в конце этой последовательности сразу осуществляется переход к коду следующей команды, поэтому требуется всего 3 цикла. Этот небольшой трюк позволяет сократить время выполнения следующей микрокоманды на один цикл, поэтому, например, последующая команда `IADD` сокращается с четырех циклов до трех. Это эквивалентно повышению частоты синхронизирующего сигнала с 250 МГц (каждая микрокоманда по 4 нс) до 333 МГц (каждая микрокоманда по 3 нс).

Таблица 4.5. Усовершенствованная микропрограмма для выполнения команды POP

Микрокоманда	Операции	Комментарий
pop1	MAR = SP = SP — 1; rd	Считывание второго сверху слова в стеке
Main1. pop	PC = PC + 1; fetch	Регистр MBR содержит код операции; вызов следующего байта
pop3	TOS = MDR; goto(MBR)	Копирование нового слова в регистр TOS; переход к коду операции

Команда POP очень хорошо подходит для такой доработки, поскольку она содержит цикл, в котором АЛУ не используется, а основной цикл требует АЛУ. Таким образом, чтобы сократить длину команды на одну микрокоманду, нужно в этой команде найти цикл, где АЛУ не используется. Такие циклы встречаются нечасто, но все-таки встречаются, поэтому включение Main1 в конце каждой последовательности микрокоманд вполне целесообразно. Для этого требуется всего лишь небольшая управляющая память. Итак, мы узнали о первой возможности сокращения длины пути:

Помещение основного цикла в конце каждой последовательности микрокоманд.

3-шинная архитектура

Что еще можно сделать для сокращения длины пути? Можно подвести к АЛУ две полные входные шины, A и B; таким образом, всего получится три шины. Все (или по крайней мере большинство регистров) должны иметь доступ к обеим входным шинам. Преимущество такой системы состоит в возможности складывать любой регистр с любым другим регистром за один цикл. Чтобы увидеть, насколько продуктивен такой подход, рассмотрим реализацию команды ILOAD (табл. 4.6).

Таблица 4.6. Микропрограмма для выполнения команды ILOAD

Микрокоманда	Операции	Комментарий
iload1	H = LV	MBR содержит индекс; копирование LV в H
iload2	MAR = MBRU + H; rd	MAR = адрес локальной переменной, которую нужно поместить в стек
iload3	MAR = SP = SP + 1	Регистр SP указывает на новую вершину стека; подготовка к записи
iload4	PC = PC + 1; fetch; wr	Увеличение PC на 1; вызов следующего кода операции; запись вершины стека
iload5	TOS = MDR; goto Main1	Обновление TOS
Main1	PC = PC + 1; fetch; goto(MBR)	Регистр MBR содержит код операции; вызов следующего байта; переход

Мы видим, что в микрокоманде iload1 значение LV копируется в регистр H. Это нужно только для того, чтобы сложить H с MBRU в микрокоманде iload2. В разработке с двумя шинами нет возможности складывать два произвольных регистра, поэтому один из них сначала нужно скопировать в регистр H. В 3-шинной архитектуре мы можем сэкономить один цикл, как показано в табл. 4.7. Мы добавили основной цикл к команде ILOAD, но при этом длина пути не увеличилась и не уменьшилась. Однако дополнительная шина сокращает общее время выполнения команды с шести циклов до пяти. Теперь мы знаем вторую возможность сокращения длины пути:

Переход от 2-шинной к 3-шинной архитектуре.

Таблица 4.7. Микропрограмма для выполнения команды ILOAD в 3-шинной архитектуре

Микро-команда	Операции	Комментарий
iload1	MAR = MBRU + LV; rd	MAR = адрес локальной переменной, которую нужно поместить в стек
iload2	MAR = SP = SP + 1	Регистр SP указывает на новую вершину стека; подготовка к записи
iload3	PC = PC + 1; fetch; wr	Увеличение PC на 1; вызов следующего кода операции; запись вершины стека
iload4	TOS = MDR	Обновление TOS
iload5	PC = PC + 1; fetch; goto(MBR)	Регистр MBR уже содержит код операции; вызов индексного байта

Блок выборки команд

Обе описанные возможности вполне применимы, но для достижения существенного эффекта потребуется нечто более радикальное. Давайте вернемся чуть-чуть назад и рассмотрим обычные составляющие любой команды: выборку и декодирование полей. Отметим, что в каждой команде могут происходить следующие операции:

- значение PC пропускается через АЛУ и увеличивается на 1;
- PC используется для вызова следующего байта в потоке команд;
- операнды считываются из памяти;
- операнды записываются в память;
- АЛУ выполняет вычисление, и результаты сохраняются в памяти.

Если команда содержит дополнительные поля (для операндов), каждое поле должно вызываться явно, по одному байту за раз. Поле можно использовать только после того, как эти байты будут объединены. При выборке и компоновке поля АЛУ должно для каждого байта увеличивать PC на единицу, а затем объединять получившийся индекс или смещение. Когда, помимо выполнения основной работы команды, приходится вызывать и объединять поля этой команды, АЛУ используется практически в каждом цикле.

Чтобы объединить основной цикл с какой-нибудь микрокомандой, нужно освободить АЛУ от некоторых задач подобного рода. Для этого можно ввести второе АЛУ, хотя полнофункциональное АЛУ в большинстве случаев для этого не потребуется. Отметим, что АЛУ часто применяется для копирования значения из одного регистра в другой. Эти циклы можно убрать, если ввести дополнительные тракты данных, которые не проходят через АЛУ. Полезно будет, например, создать тракт от TOS к MDR или от MDR к TOS, поскольку верхнее слово стека часто копируется из одного регистра в другой.

В микроархитектуре Mic-1 с АЛУ можно снять большую часть нагрузки, если создать независимый блок для вызова и обработки команд. Этот блок, который называется **блоком выборки команд** (Instruction Fetch Unit, **IFU**), может независимо от АЛУ увеличивать значение PC на единицу и вызывать байты из потока байтов до того, как они понадобятся. Блок IFU содержит схему инкремента, которая по строению гораздо проще, чем полный сумматор. Разовьем эту идею.

Блок выборки команд может также объединять 8-разрядные и 16-разрядные операнды, чтобы они могли использоваться сразу, как только потребуются. Это можно осуществить по крайней мере двумя способами:

- Блок IFU может интерпретировать каждый код операции, определять, сколько дополнительных полей нужно вызвать, и собирать их в регистр, который будет использоваться основным операционным блоком.
- Блок IFU может постоянно предоставлять следующие 8- или 16-разрядные фрагменты данных независимо от того, имеет это смысл или нет. Тогда основной операционный блок может запрашивать любые данные, которые ему требуются.

Рудиментарная реализация второго способа представлена на рис. 4.18. Вместо одного 8-разрядного регистра MBR присутствуют два регистра MBR: 8-разрядный MBR1 и 16-разрядный MBR2. Блок IFU следит за самым последним байтом или байтами, которые поступили в основной операционный блок. Кроме того, он передает следующий байт в регистр MBR, как и в архитектуре Mic-1, только в данном случае он автоматически определяет, когда значение регистра считано, вызывает следующий байт и сразу загружает его в регистр MBR1. Как и в микроархитектуре Mic-1, он имеет два интерфейса с шиной B: MBR1 и MBR1U. Первый получает знаковое расширение до 32 битов, второй дополнен нулями.

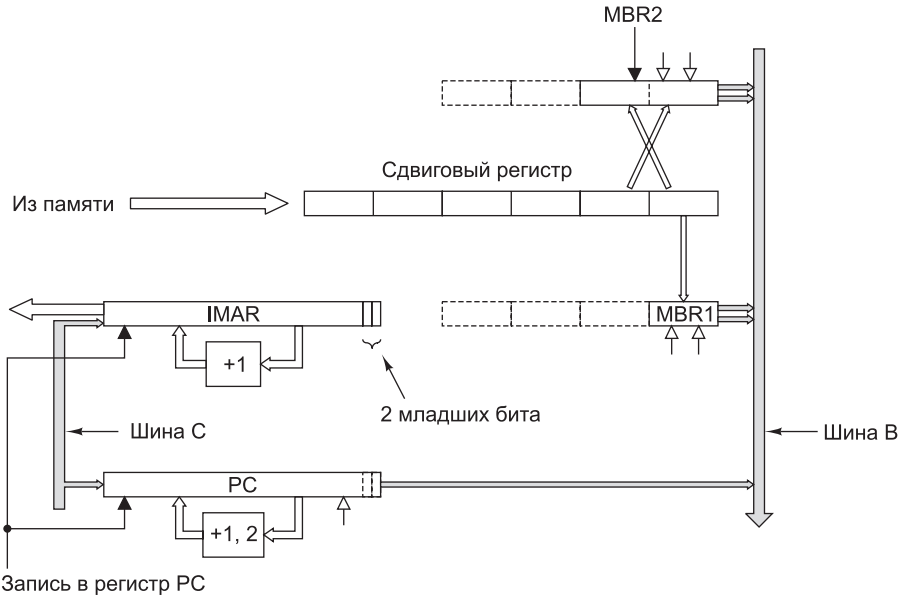


Рис. 4.18. Блок выборки команд в микроархитектуре Mic-1

Регистр MBR2 функционирует точно так же, но содержит следующие 2 байта. Он имеет два интерфейса с шиной B: MBR2 и MBR2U, первый из которых расширен по знаку, а второй дополнен до 32 бит нулями.

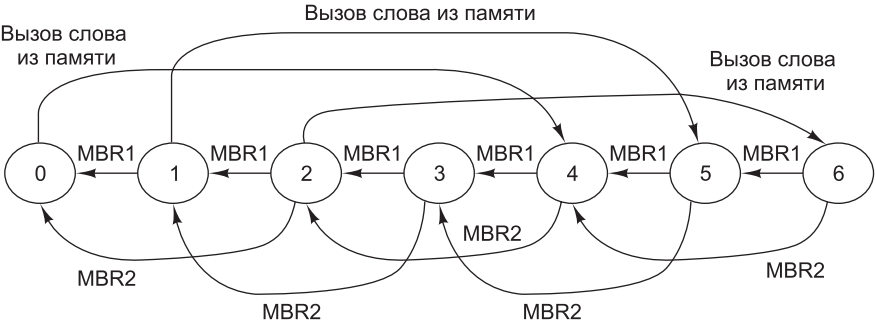
Блок выборки команд отвечает за выборку потока байтов. Для этого он использует стандартный 4-байтный порт памяти, вызывая полные 4-байтные слова заранее и загружая следующие байты в сдвиговый регистр, который выдает их по одному или по два за раз в том порядке, в котором они вызываются из памяти.

Задача сдвигового регистра — сохранить последовательность поступающих байтов для загрузки в регистры MBR1 и MBR2.

MBR1 всегда содержит самый старший байт сдвигового регистра, а MBR2 — 2 старших байта (старшим является левый байт), которые формируют 16-разрядное целое число (см. рис. 4.14, *б*). Два байта в регистре MBR2 могут быть получены из различных слов памяти, поскольку IJVM-команды никак не связаны с границами слов.

Всякий раз, когда считывается регистр MBR1, значение сдвигового регистра сдвигается вправо на 1 байт. При считывании регистра MBR2 значение сдвигового регистра сдвигается вправо на 2 байта. Затем в регистры MBR1 и MBR2 загружаются самый старший байт и пара самых старших байтов соответственно. Если к этому моменту в сдвиговом регистре остается достаточно места для целого слова, блок выборки команд начинает цикл обращения к памяти, чтобы считать следующее слово. Предполагается, что при считывании любого из регистров MBR он заново заполняется к началу следующего цикла, поэтому новое значение можно считывать уже в последующих циклах.

Блок выборки команд может быть смоделирован в виде **конечного автомата** (Finite State Machine, **FSM**), как показано на рис. 4.19. Любой конечный автомат характеризуется **состояниями** (на рисунке это кружочки) и **переходами** (это дуги от одного состояния к другому). Каждое состояние — это одна из возможных ситуаций, в которой может находиться конечный автомат. Данный конечный автомат имеет семь состояний, которые соответствуют семи состояниям сдвигового регистра, показанного на рис. 4.18. Эти семь состояний соответствуют количеству байтов, которые находятся в данный момент в регистре (от 0 до 6 включительно).



Переходы

MBR1: Происходит при чтении MBR1

MBR2: Происходит при чтении MBR2

Вызов слова: Происходит при считывании слова из памяти и помещении 4 байтов в регистр сдвига

Рис. 4.19. Конечный автомат для реализации блока выборки команд

Каждая дуга отражает возможное событие. В нашем конечном автомате возможны три различных события. Первое — чтение одного байта из регистра MBR1. Оно активизирует сдвиговый регистр, самый правый байт в нем исчезает, и осуществляется переход в другое состояние (меньшее на 1). Второе событие — чтение 2 байт из регистра MBR2. При этом осуществляется переход в состояние, меньшее на 2 (например, из состояния 2 в состояние 0 или из со-

стояния 5 в состояние 3). Оба этих перехода вызывают перезагрузку регистров MBR1 и MBR2. Когда конечный автомат переходит в состояния 0, 1 или 2, инициируется обращение к памяти, чтобы вызвать новое слово (предполагается, что память уже не занята считыванием предыдущего слова). При поступлении слова номер состояния увеличивается на 4.

Чтобы схема выборки команд функционировала правильно, она должна блокироваться в том случае, если от нее требуют произвести какие-то действия, которые она выполнить не может (например, передать значение в MBR2, когда в сдвиговом регистре находится только 1 байт, а память все еще занята вызовом нового слова). Кроме того, блок выборки команд не может выполнять несколько операций одновременно, поэтому все входящие события должны передаваться последовательно. Наконец, при каждом изменении PC блок выборки команд должен обновляться. Все эти детали усложняют работу блока. Однако многие устройства разрабатываются в виде конечных автоматов.

Блок выборки команд имеет собственный регистр адреса ячейки памяти, называемый IMAR и используемый для обращения к памяти, когда нужно вызвать новое слово. У этого регистра есть специальная схема инкремента, поэтому основному АЛУ не требуется прибавлять единицу к значению PC для вызова следующего слова. Блок выборки команд должен контролировать шину C, чтобы каждый раз при загрузке регистра PC новое значение PC также копировалось в IMAR. Поскольку новое значение в регистре PC может быть не на границе слова, блок выборки команд должен вызвать нужное слово и скорректировать значение сдвигового регистра соответствующим образом.

Основной операционный блок записывает значение в PC только в том случае, если необходимо изменить характер последовательности байтов. Это происходит в команде перехода, а также в командах `INVOKEVIRTUAL` и `IRETURN`.

Поскольку микропрограмма больше не увеличивает PC явным образом при вызове кода операции, блок выборки команд должен обновлять PC сам. Как это происходит? Блок IFU способен распознать, что байт из потока команд получен, то есть что значения регистров MBR1 и MBR2 (или их вариантов без знака) уже считаны. С регистром PC связана отдельная схема инкремента, которая увеличивает значение на 1 или на 2 в зависимости от того, сколько байтов получено. Таким образом, регистр PC всегда содержит адрес первого еще не полученного байта. В начале каждой команды в регистре MBR находится адрес кода операции этой команды.

Отметим, что существует две разных схемы инкремента, которые выполняют разные функции. Регистр PC считает *байты* и увеличивает значение на 1 или на 2. Регистр IMAR считает *слова* и увеличивает значение только на 1 (для четырех новых байтов). Как и MAR, регистр IMAR соединен с адресной шиной «по диагонали»: бит 0 регистра IMAR связан с адресной линией 2 и т. д. для выполнения неявного перехода от адреса слова к адресу байта.

Мы скоро увидим, что если нет необходимости увеличивать значение PC в основном цикле, это дает большой выигрыш, поскольку обычно в микрокоманде, в которой происходит увеличение PC, помимо этого больше ничего не происходит. Если эту команду устранить, длина пути сократится. Однако для увеличения скорости работы машины требуется больше аппаратного обеспечения. Таким образом, мы пришли к третьей возможности сокращения длины пути:

Выборка команд из памяти осуществляется специализированным функциональным блоком.

Упреждающая выборка команд из памяти — микроархитектура Mic-2

Блок выборки команд может значительно сократить длину пути для средней команды. Во-первых, он полностью устраняет основной цикл, поскольку в конце каждой команды просто сразу осуществляется переход к следующей. Во-вторых, АЛУ не нужно увеличивать значение PC. В-третьих, блок IFU сокращает длину пути всякий раз, когда вычисляется 16-разрядный индекс или смещение, поскольку объ-

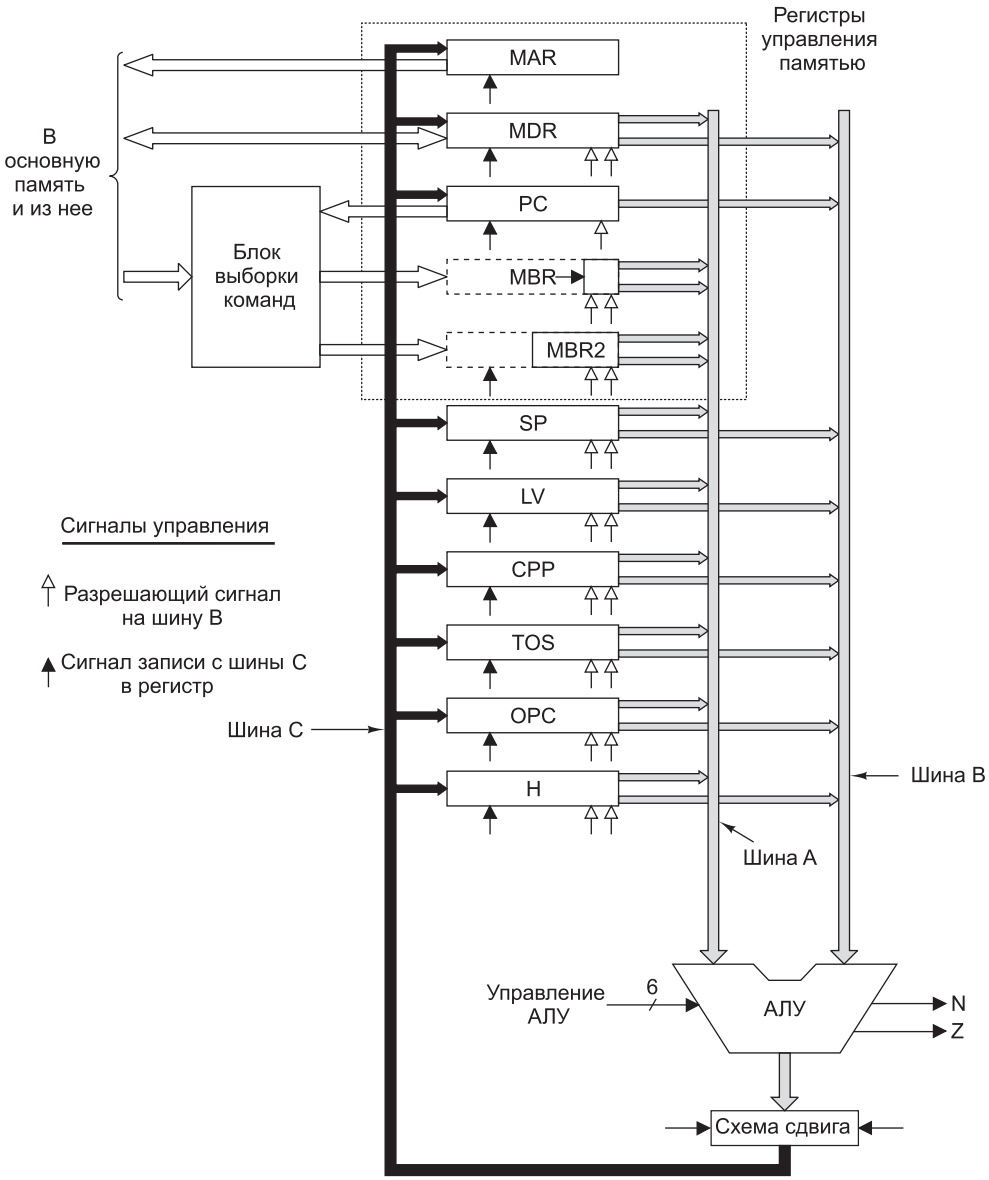


Рис. 4.20. Тракт данных для Mic-2

единяет 16-разрядное значение и сразу передает его в АЛУ в виде 32-разрядного значения без необходимости производить объединение в регистре H. На рис. 4.20 показана микроархитектура Mic-2 — усовершенствованная версия Mic-1, к которой добавлен блок выборки команд, изображенный на рис. 4.18. Микропрограмма для усовершенствованной машины приведена в табл. 4.8.

Таблица 4.8. Микропрограмма для Mic-2

Микро-команда	Операции	Комментарий
nop1	goto (MBR)	Переход к следующей команде
iadd1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
iadd2	H = TOS	H = вершина стека
iadd3	MDR = TOS = MDR + H; wr; goto (MBR1)	Суммирование двух верхних слов; запись суммы в верхнюю позицию стека
isub1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
isub2	H = TOS	H = вершина стека
isub3	MDR = TOS = MDR — H; wr; goto (MBR1)	Вычитание TOS из вызванного значения TOS — 1
iand1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
iand2	H = TOS	H = вершина стека
iand3	MDR = TOS = MDR И H; wr; goto (MBR1)	Логическое умножение вызванного значения TOS — 1 и TOS (операция И)
ior1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
ior2	H = TOS	H = вершина стека
ior3	MDR = TOS = MDR ИЛИ H; wr; goto (MBR1)	Логическое сложение вызванного значения TOS — 1 и TOS (операция ИЛИ)
dup1	MAR = SP = SP + 1	Увеличение SP на 1 и копирование результата в регистр MAR
dup2	MDR = TOS; wr; goto (MBR1)	Запись нового слова в стек
pop1	MAR = SP = SP — 1; rd	Чтение слова, идущего после верхнего слова стека
pop2		Программа ждет, пока закончится процесс чтения
pop3	TOS = MDR; goto (MBR1)	Копирование нового слова в регистр TOS
swap1	MAR = SP — 1; rd	Чтение второго слова из стека; установка регистра MAR на значение SP

продолжение ⇩

Таблица 4.8 *(продолжение)*

Микро-команда	Операции	Комментарий
swap2	MAR = SP	Подготовка к записи нового второго слова стека
swap3	H = MDR; wr	Сохранение нового значения TOS; запись второго слова стека
swap4	MDR = TOS	Копирование прежнего значения TOS в регистр MDR
swap5	MAR = SP — 1; wr	Запись прежнего значения TOS на второе место в стеке
swap6	TOS = H; goto (MBR1)	Обновление TOS
bipush1	SP = MAR = SP + 1	Установка регистра MAR для записи в новую вершину стека
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Обновление стека в регистре TOS и памяти
iload1	MAR = LV + MBR1U; rd	Перемещение значения LV с индексом в регистр MAR; чтение операнда
iload2	MAR = SP = SP + 1	Увеличение SP на 1; перемещение нового значения SP в регистр MAR
iload3	TOS = MDR; wr; goto (MBR1)	Обновление стека в регистре TOS и памяти
istore1	MAR = LV + MBR1U	Установка регистра MAR на значение LV + индекс
istore2	MDR = TOS; wr	Копирование значения TOS для сохранения в памяти
istore3	MAR = SP = SP — 1; rd	Уменьшение SP на 1; чтение нового значения TOS
istore4		Машина ждет окончания процесса чтения
istore5	TOS = MDR; goto (MBR1)	Обновление TOS
wide1	goto (MBR1 ИЛИ 0x100)	Следующий адрес — 0x100 ИЛИ код операции
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	То же что iload1, но с использованием 2-байтного индекса
wide_istore1	MAR = LV + MBR2U; goto istore2	То же что istore1, но с использованием 2-байтного индекса
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	То же что wide_iload1, но индексирование осуществляется из регистра CPP
iinc1	MAR = LV + MBR1U; rd	Установка регистра MAR на значение LV + индекс; чтение этого значения

Микро-команда	Операции	Комментарий
iinc2	H = MBR1	Присваивание регистру H константы
iinc3	MDR = MDR + H; wr; goto (MBR1)	Увеличение на константу и обновление
goto1	H = PC — 1	Копирование значения PC в регистр H
goto2	PC = H + MBR2	Прибавление смещения и обновление PC
goto3		Машина ждет, пока блок выборки команд вызовет новый код операции
goto4	goto (MBR1)	Переход к следующей команде
iflt1	MAR = SP = SP — 1; rd	Чтение второго сверху слова в стеке
iflt2	OPC = TOS	Временно́е сохранение TOS в OPC
iflt3	TOS = MDR	Запись новой вершины стека в TOS
iflt4	N = OPC; if(N) goto T; else goto F	Переход по биту N
ifeq1	MAR = SP = SP — 1; rd	Чтение второго сверху слова в стеке
ifeq2	OPC = TOS	Временно́е сохранение TOS в OPC
ifeq3	TOS = MDR	Запись новой вершины стека в TOS
ifeq4	Z = OPC; if(Z) goto T; else goto F	Переход по биту Z
if_icmpeq1	MAR = SP = SP — 1; rd	Чтение второго сверху слова в стеке
if_icmpeq2	MAR = SP = SP — 1	Установка регистра MAR на чтение новой вершины стека
if_icmpeq3	H = MDR; rd	Копирование второго слова из стека в регистр H
if_icmpeq4	OPC = TOS	Временно́е сохранение TOS в OPC
if_icmpeq5	TOS = MDR	Помещение новой вершины стека в TOS
if_icmpeq6	Z = H — OPC; if(Z) goto T; else goto F	Если два верхних слова равны, осуществляется переход к T; если они не равны, осуществляется переход к F
T	H = PC — 1; goto goto2	То же, что goto1
F	H = MBR2	Игнорирование байтов, находящихся в регистре MBR2
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	Помещение адреса указателя процедуры в регистр MAR
invokevirtual2	OPC = PC	Сохранение значения PC в регистре OPC

продолжение ⇩

Таблица 4.8 *(продолжение)*

Микро-команда	Операции	Комментарий
invokevirtual3	PC = MDR	Установка PC на первый байт кода процедуры
invokevirtual4	TOS = SP – MBR2U	TOS = адрес OBJREF – 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = адрес OBJREF
invokevirtual6	MDR = SP + MBR2U + 1; wr	Перезапись OBJREF со связующим указателем
invokevirtual7	MAR = SP = MDR	Установка регистров SP и MAR на адрес ячейки, в которой содержится старое значение PC
invokevirtual8	MDR = OPC; wr	Подготовка к сохранению старого значения PC
invokevirtual9	MAR = SP = SP + 1	Увеличение SP на 1; теперь SP указывает на ячейку, в которой хранится старое значение LV
invokevirtual10	MDR = LV; wr	Сохранение старого значения LV
invokevirtual11	LV = TOS; goto (MBR1)	Установка значения LV на нулевой параметр
ireturn1	MAR = SP = LV; rd	Переустановка регистров SP и MAR для чтения связующего указателя
ireturn2		Процесс считывания связующего указателя
ireturn3	LV = MAR = MDR; rd	Установка регистров LV и MAR на связующий указатель; чтение старого значения PC
ireturn4	MAR = LV + 1	Установка регистра MAR на старое значение LV; чтение старого значения LV
ireturn5	PC = MDR; rd	Восстановление PC
ireturn6	MAR = SP	
ireturn7	LV = MDR	Восстановление LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Сохранение результата в иисходной вершине стека

Чтобы продемонстрировать, как работает Mic-2, рассмотрим команду `IADD`. Она берет второе слово из стека и выполняет сложение как и раньше, но только сейчас ей не нужно осуществлять переход к `Main1` после завершения операции, чтобы увеличить значение PC и перейти к следующей микрокоманде. Когда блок выборки команд распознает, что в цикле `iadd3` произошло обращение к регистру `MBR1`, его внутренний сдвиговый регистр сдвигает все вправо и переза-

гружает MBR1 и MBR2. Он также осуществляет переход в состояние, которое на единицу меньше текущего. Если новое состояние — это состояние 2, блок выборки команд начинает вызов слова из памяти. Все это реализуется аппаратно. Микропрограмма ничего не должна делать. Именно поэтому команду `IADD` можно сократить с четырех до трех микрокоманд.

В микроархитектуре Mic-2 некоторые команды усовершенствованы в большей степени, чем другие. Команда `LDC_W` сокращается с 9 до 3 микрокоманд, и, следовательно, время выполнение команды уменьшается втрое. С другой стороны, команда `SWAP` изначально содержала 8 микрокоманд, а стало 6. Для общей производительности компьютера играет роль сокращение наиболее часто повторяющихся команд, а это команды `ILOAD` (было 6, стало 3), `IADD` (было 4, стало 3) и `IF_ICMPEQ` (было 13, стало 10 для случая, когда слова равны; было 10, стало 8 для случая, когда слова не равны). Чтобы вычислить, насколько выросла производительность, можно проверить эффективность системы по эталонному тесту, но и без этого ясно, что имеет место значительный выигрыш в быстродействии.

Конвейерная конструкция — микроархитектура Mic-3

Очевидно, что микроархитектура Mic-2 совершеннее Mic-1. Она работает быстрее и требует меньше управляющей памяти, хотя стоимость блока выборки команд несомненно превышает сумму, выигранную за счет сокращения объема управляющей памяти. Таким образом, машина Mic-2 работает значительно быстрее при некотором увеличении стоимости. Давайте посмотрим, можно ли еще больше повысить скорость.

А что если попробовать уменьшить время цикла? В значительной степени время цикла определяется базовой технологией. Чем меньше транзисторы и физическое расстояние между ними, тем быстрее может работать задающий генератор. В технологии, которую мы рассматриваем, время, затрачиваемое на прохождение через тракт данных, фиксировано (по крайней мере, с нашей точки зрения). Тем не менее у нас есть некоторая свобода действий и далее мы используем ее в полной мере.

Еще один вариант усовершенствования — увеличить степень параллелизма. На данный момент микроархитектура Mic-2 выполняет большинство операций последовательно. Она помещает значения регистров на шины, ждет, пока АЛУ и схема сдвига их обработают, а затем записывает результаты обратно в регистры. Если не учитывать работу блока выборки команд, никакого параллелизма здесь нет. Внедрение дополнительных механизмов параллельной обработки сулит значительные преимущества.

Как уже отмечалось, длительность цикла определяется временем, необходимым для прохождения сигнала через тракт данных. На рис. 4.2 показано распределение этой задержки между различными компонентами во время каждого цикла. Цикл тракта данных объединяет три основных составляющих:

1. Время, которое требуется на передачу значений выбранных регистров на шины A и B.
2. Время, которое требуется на работу АЛУ и схемы сдвига.
3. Время, которое требуется на передачу полученных значений обратно в регистры и сохранение этих значений.

На рис. 4.21 показана новая 3-шинная архитектура с блоком выборки команд и тремя дополнительными защелками (регистрами), каждая из которых расположена в середине каждой шины. Эти регистры записываются в каждом цикле. Они делят тракт данных на отдельные части, которые могут функционировать независимо друг от друга. Мы будем называть такую архитектуру **конвейерной** моделью, или **Mic-3**.

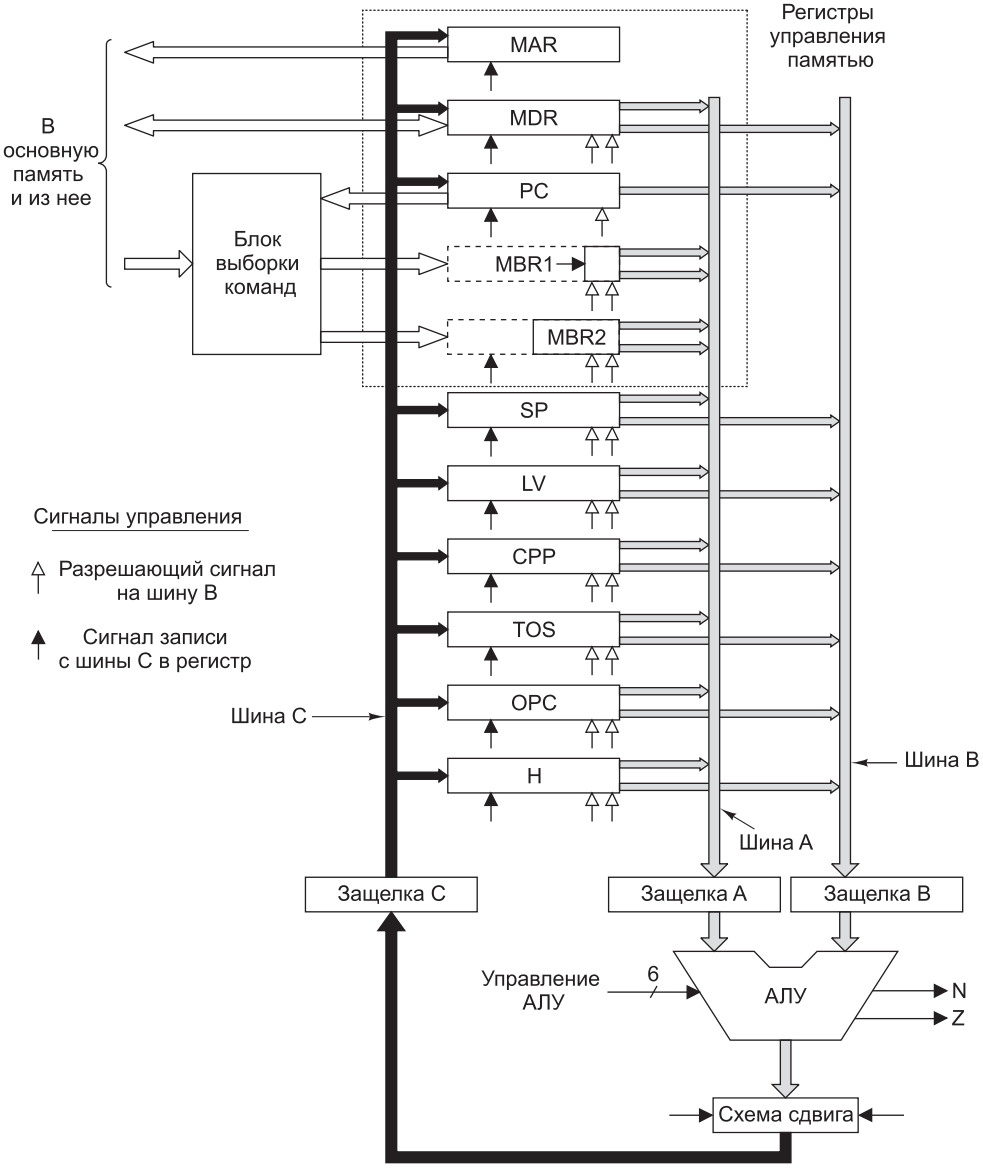


Рис. 4.21. Тракт данных с тремя шинами в микроархитектуре Mic-3

Зачем нужны целых три дополнительных регистра? Ведь теперь для прохождения сигнала через тракт данных требуются 3 цикла: один для загрузки регистров A и B, второй для запуска АЛУ и схемы сдвига, а также загрузки регистра C, третий для сохранения значения регистра-защелки C обратно в нужных регистрах. Мы что, ненормальные? (*Подсказка*: нет.) Существует целых две причины введения дополнительных регистров:

1. Мы можем повысить тактовую частоту, поскольку максимальная задержка теперь стала меньше.
2. Во время каждого цикла мы можем использовать все части тракта данных.

После разбиения тракта данных на три части максимальная задержка прохождения сигнала уменьшается, в результате тактовая частота может повышаться. Будем считать, что если разбить цикл тракта данных на три примерно равных интервала, тактовая частота увеличится втрое. (На самом деле это не так, поскольку мы добавили в тракт данных еще два регистра, но в первом приближении это допустимо.)

Поскольку мы предполагаем, что все операции чтения из памяти и записи в память выполняются с использованием кэш-памяти первого уровня и эта кэш-память построена из того же материала, что и регистры, мы можем сделать вывод, что операция с памятью занимает один цикл. На практике, однако, этого не так легко достичь.

Второй пункт связан с общей производительностью, а не со скоростью выполнения отдельной команды. В микроархитектуре Mic-2 во время первой и третьей частей каждого цикла АЛУ простаивает. Если разделить тракт данных на три части, то появляется возможность использовать АЛУ в каждом цикле, вследствие чего производительность машины увеличивается втрое.

А теперь посмотрим, как работает тракт данных Mic-3. Однако сначала нужно как-то назвать защелки. Проще всего назвать защелки A, B и C и считать их регистрами, подразумевая ограничения тракта данных. В табл. 4.9 приведен фрагмент программы для микроархитектуры Mic-2 (реализация команды `SWAP`).

Таблица 4.9. Программа Mic-2 для команды SWAP

Микрокоманда	Операции	Комментарий
swap1	MAR = SP — 1; rd	Чтение второго слова из стека; установка MAR на SP
swap2	MAR = SP	Подготовка к записи нового второго слова
swap3	H = MDR; wr	Сохранение нового значения TOS; запись второго слова в стек
swap4	MDR = TOS	Копирование старого значения TOS в регистр MDR
swap5	MAR = SP — 1; wr	Запись старого значения TOS на второе место в стеке
swap6	TOS = H; goto (MBR1)	Обновление TOS

Давайте перепишем эту последовательность для Mic-3. Следует помнить, что теперь тракт данных работает три цикла: один служит для загрузки регистров A

и B, второй — для выполнения операции и загрузки регистра C, третий — для записи результатов в регистры. Каждый из этих циклов мы назовем **микрошагом**.

Реализация команды `SWAP` для Mic-3 показана в табл. 4.10. В цикле 1 мы начинаем микрокоманду `swap1`, копируя значение SP в регистр B. Не имеет никакого значения, что происходит в регистре A, поскольку чтобы отнять 1 из B, ENA (сигнал разрешения A) блокируется (см. табл. 4.1). Для простоты мы не показываем присваивания, которые не используются. В цикле 2 производится операция вычитания. В цикле 3 результат сохраняется в регистре MAR и после этого, в конце третьего цикла, начинается процесс чтения. Поскольку чтение из памяти занимает один цикл, закончится он только в конце четвертого цикла. Это показано присваиванием значения регистру MDR в цикле 4. Значение из MDR можно считывать не раньше пятого цикла.

Таблица 4.10. Реализация команды SWAP в архитектуре Mic-3

	Swap1	**Swap2**	**Swap3**	**Swap4**	**Swap5**	**Swap6**
Цикл	MAR = SP — 1; rd	MAR = SP	H = MDR; wr	MDR = TOS	MAR = SP — 1; wr	TOS = H; goto (MBR1)
1	B = SP					
2	C = B — 1	B = SP				
3	MAR = C; rd	C = B				
4	MDR = Mem	MAR = C				
5			B = MDR			
6			C = B	B = TOS		
7			H = C; wr	C = B	B = SP	
8			Mem = MDR	MDR = C	C = B — 1	B = H
9					MAR = C; wr	C=B
10					Mem = MDR	TOS = C
11						goto (MBR1)

А теперь вернемся к циклу 2. Мы можем разбить микрокоманду `swap2` на микрошаги и начать их выполнение. В цикле 2 мы копируем значение SP в регистр B, затем пропускаем значение через АЛУ в цикле 3 и, наконец, сохраняем его в регистре MAR в цикле 4. Пока все хорошо. Должно быть ясно, что если мы сможем начинать новую микрокоманду в каждом цикле, скорость работы машины увеличится в три раза. Такое повышение скорости происходит за счет того, что машина Mic-3 производит в три раза больше циклов в секунду, чем Mic-2. Фактически, мы построили конвейерный процессор.

К сожалению, в цикле 3 возникает препятствие. Прекрасно было бы начать микрокоманду `swap3`, но эта микрокоманда сначала пропускает значение MDR через АЛУ, а значение MDR не может быть получено из памяти до начала цикла 5. Ситуация, когда следующий микрошаг не может начаться, потому что перед этим нужно получить результат выполнения предыдущего микрошага, называ-

ется **реальной взаимозависимостью**, или **RAW-взаимозависимостью** (Read After Write — **чтение после записи**). Взаимозависимости иногда называют **рисками**. В такой ситуации требуется считать значение регистра, которое еще не записано. Единственное разумное решение в данном случае — отложить начало микрокоманды `swap3` до того момента, когда значение MDR станет доступным, то есть до пятого цикла. Ожидание нужного значения называется **простоем**. После этого мы можем начинать выполнение микрокоманд в каждом цикле, поскольку таких ситуаций больше не возникает, хотя имеется пограничная ситуация: микрокоманда `swap6` считывает значение регистра H в цикле, который следует сразу после записи этого регистра в микрокоманде `swap3`. Если бы значение этого регистра считывалось в микрокоманде `swap5`, машине пришлось бы простаивать один цикл.

Хотя программа Mic-3 требует больше циклов, чем программа Mic-2, она работает гораздо быстрее. Если время цикла микроархитектуры Mic-3 составляет ΔT нс, то для выполнения команды `SWAP` машине Mic-3 требуется $11\Delta T$ нс, а машине Mic-2 — 6 циклов по $3\Delta T$ нс каждый, то есть всего $18\Delta T$ нс. Конвейеризация увеличивает быстродействие компьютера даже несмотря на то, что один раз приходится простаивать из-за явления реальной взаимозависимости.

Конвейеризация является ключевой технологией во всех современных процессорах, поэтому важно хорошо в ней разбираться. На рис. 4.22 графически проиллюстрирована конвейеризация тракта данных, изображенного на рис. 4.21. В первой колонке показано, что происходит во время цикла 1, вторая колонка представляет цикл 2 и т. д. (предполагается, что простоев нет). Закрашенная область на рисунке для цикла 1 и команды 1 означает, что блок выборки команд занят вызовом команды 1. В цикле 2 значения регистров, вызванных командой 1, загружаются в A и B, а в это время блок выборки команд занимается вызовом команды 2, что также показано закрашенными серыми прямоугольниками.

Во время цикла 3 команда 1 использует АЛУ и схему сдвига, регистры A и B загружаются для команды 2, вызывается команда 3. Наконец, во время цикла 4 работают все 4 команды одновременно. Сохраняются результаты выполнения команды 1, АЛУ выполняет вычисления для команды 2, регистры A и B загружаются для команды 3, вызывается команда 4.

Если бы мы показали цикл 5 и следующие, модель была бы точно такой же, как в цикле 4: все четыре части тракта данных работали бы независимо друг от друга. Данный конвейер содержит 4 ступени: для вызова команд, для доступа к операндам, для работы АЛУ и для записи результата обратно в регистры. Он похож на конвейер, изображенный на рис. 2.3, *a*, только у него отсутствует ступень декодирования (расшифровки). Здесь важно подчеркнуть, что хотя выполнение одной команды занимает 4 цикла, в каждом цикле начинается новая команда и завершается предыдущая.

Можно рассматривать схему на рис. 4.22 не вертикально (по колонкам), а горизонтально (по рядам). При выполнении команды 1 в цикле 1 функционирует блок выборки команд. В цикле 2 значения регистров помещаются на шины A и B. В цикле 3 работают АЛУ и схема сдвига. Наконец, в цикле 4 полученные результаты сохраняются в регистрах. Отметим, что имеется 4 доступных устройства, и во время каждого цикла определенная команда использует только одно из них, оставляя свободными другие устройства для других команд.

Проведем аналогию с конвейером на заводе по производству автомобилей. Чтобы изложить суть работы такого конвейера, представим, что ровно каждую

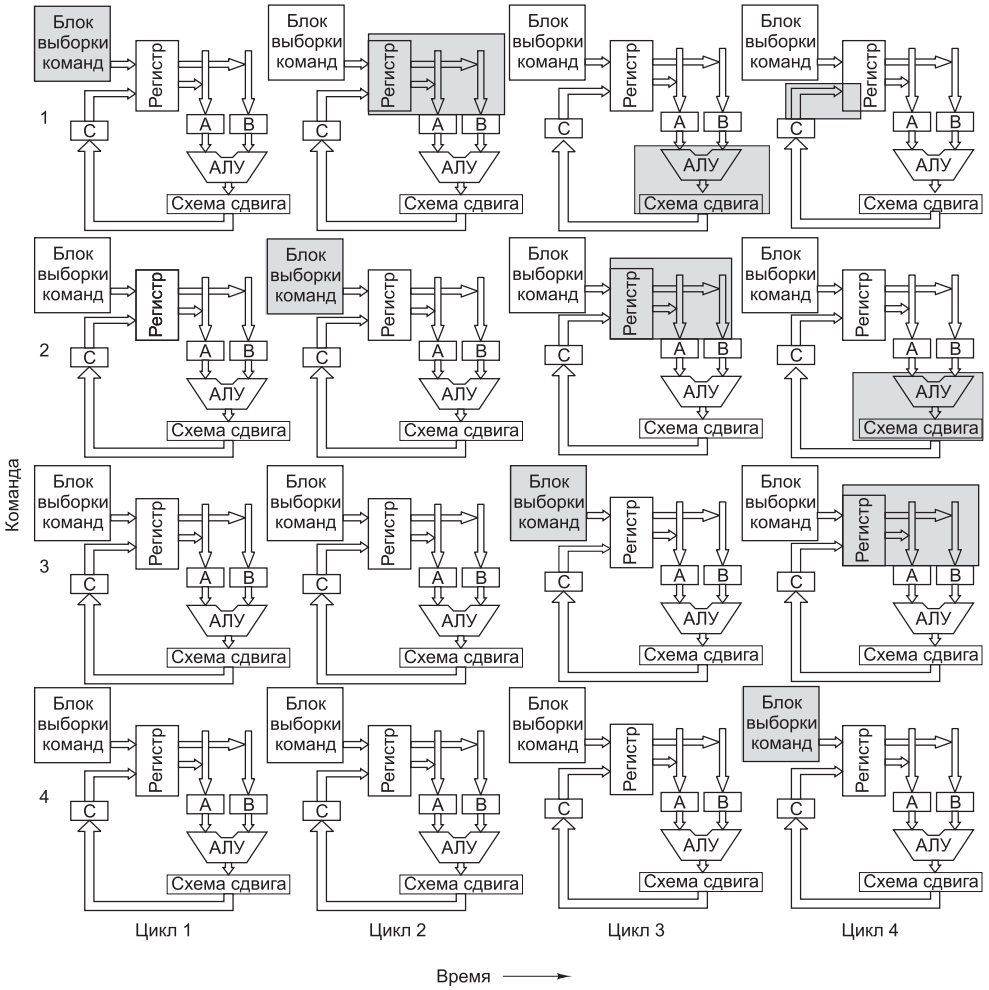


Рис. 4.22. Графическая иллюстрация работы конвейера

минуту звучит гонг, и в этот момент все автомобили передвигаются по конвейеру на один шаг. На каждом шаге рабочие выполняют определенную операцию с автомобилем, который оказывается перед ними, например ставят колеса или тормоза. При каждом ударе гонга (это — 1 цикл) очередная заготовка поступает на конвейер и один собранный автомобиль сходит с конвейера. Таким образом, завод выпускает один автомобиль в минуту независимо от того, сколько времени занимает сборка одного автомобиля. В этом и состоит суть конвейера. Такой подход в равной степени применим и к процессорам, и к автомобилям.

Семиступенчатый конвейер — микроархитектура Mic-4

Мы обошли вниманием тот факт, что каждая микрокоманда выбирает следующую за ней. Большинство из них просто выбирают очередную команду в текущей

последовательности, но последняя из них, например swap6, часто совершает межуровневый переход, который останавливает работу конвейера, поскольку после этого перехода вызывать команды заранее уже бессмысленно. Необходимо найти более удачное решение этой проблемы.

Следующая (и последняя) микроархитектура — Mic-4. Ее основные компоненты представлены на рис. 4.23, но большая их часть не показана, чтобы сделать схему более понятной. Как и Mic-3, эта микроархитектура содержит блок выборки команд (IFU), который заранее вызывает слова из памяти и сохраняет различные значения MBR.

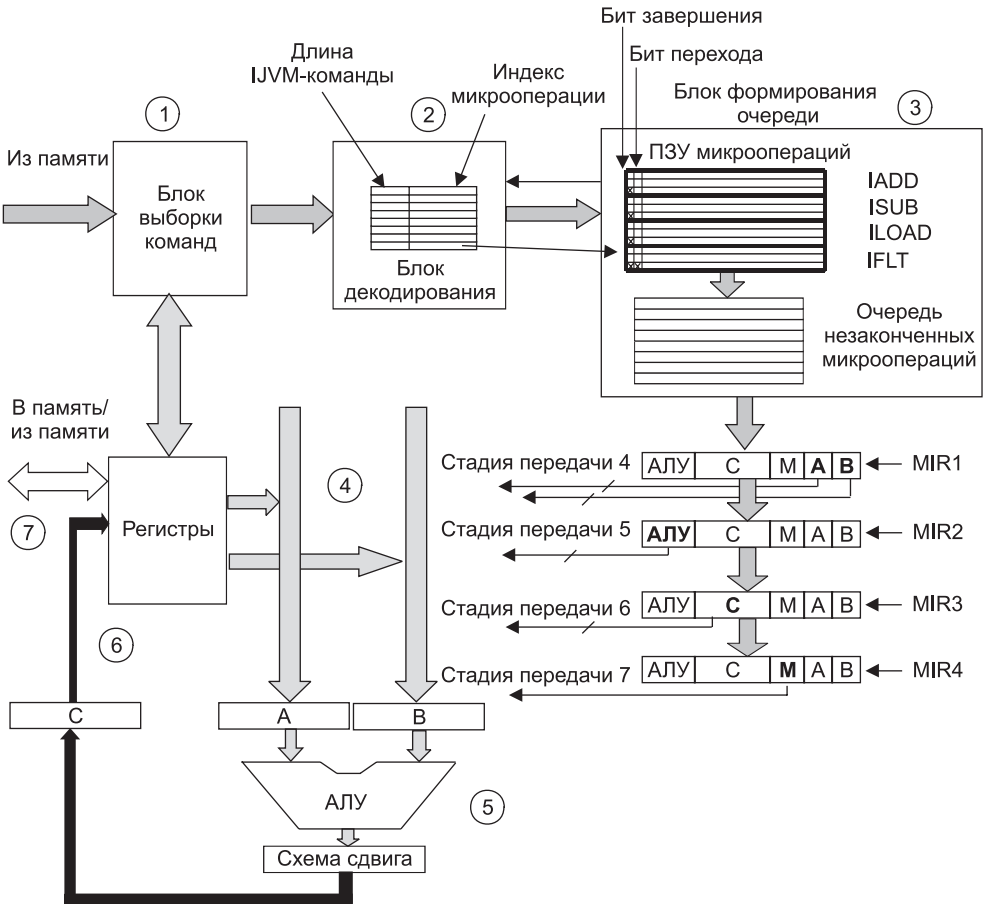


Рис. 4.23. Основные компоненты микроархитектуры Mic-4

Блок выборки команд передает входящий поток байтов в новый компонент — **блок декодирования**. Этот блок содержит внутреннее ПЗУ, которое индексируется кодом IJVM-операции. Каждый элемент (ряд) блока состоит из двух частей: поля длины IJVM-команды и индекса в другом ПЗУ — ПЗУ микроопераций. Длина IJVM-команды нужна для того, чтобы блок декодирования мог разделить входящий поток байтов и установить, какие байты являются кодами операций,

а какие — операндами. Если длина текущей команды составляет 1 байт (например, длина команды `POP`), то блок декодирования определяет, что следующий байт — это код операции. Если длина текущей команды составляет 2 байта, блок декодирования определяет, что следующий байт — это операнд, сразу за которым следует другой код операции. Когда появляется префиксная команда `WIDE`, следующий байт преобразуется в специальный расширенный код операции; например, `WIDE` плюс `ILOAD` превращается в `WIDE_ILOAD`.

Блок декодирования передает индекс в ПЗУ микроопераций, который он находит в своей таблице, следующему компоненту, **блоку формирования очереди**. Этот блок содержит логические схемы и две внутренние таблицы, одну для ПЗУ и вторую для ОЗУ. В ПЗУ находится микропрограмма, причем каждая IJVM-команда состоит из **микроопераций**. Эти микрооперации должны располагаться в строгом порядке, и, например, переход из `wide_iload2` в `iload2`, который допустим в микроархитектуре Mic-2, не разрешается. Каждая последовательность микроопераций должна выполняться полностью, в некоторых случаях последовательности дублируются.

Структура микрооперации сходна со структурой микрокоманды (см. рис. 4.4), только в данном случае поля NEXT_ADDRESS и JAM отсутствуют, и требуется новое поле для определения ввода с шины A. Имеются также два новых бита: бит завершения и бит перехода. Бит завершения устанавливается на последней микрооперации каждой последовательности (чтобы обозначить эту операцию). Бит перехода нужен для указания на микрооперации, которые являются условными микропереходами. По формату они отличаются от обычных микроопераций. Они состоят из битов JAM и индекса в ПЗУ микроопераций. Микрокоманды, которые раньше осуществляли какие-либо действия с трактом данных, а также выполняли условные микропереходы (например, `iflt4`), теперь приходится разбивать на две микрооперации.

Блок формирования очереди получает от блока декодирования индекс микрооперации в ПЗУ микроопераций. Затем он отыскивает микрооперацию и копирует ее во внутреннюю очередь. После этого он копирует очередную микрооперацию в ту же очередь, а также микрооперацию, следующую за этой микрооперацией. Так продолжается до тех пор, пока не появится микрооперация с битом завершения. Тогда блок копирует эту последнюю микрооперацию и останавливается. Если блоку не встретилась микрооперация с битом перехода и у него осталось достаточно свободного пространства, он посылает сигнал подтверждения приема блоку декодирования. Когда блок декодирования принимает сигнал подтверждения, он посылает блоку формирования очереди следующую IJVM-команду.

Таким образом, последовательность IJVM-команд в памяти в конечном итоге превращается в последовательность микроопераций в очереди. Эти микрооперации передаются в регистры MIR, которые посылают сигналы управления трактом данных. Но есть еще один фактор, который нам нужно учесть: поля каждой микрооперации не действуют одновременно. Поля A и B активны во время первого цикла, поле АЛУ активно во время второго цикла, поле C активно во время третьего цикла, а все операции с памятью происходят в четвертом цикле.

Чтобы все эти операции выполнялись правильно, мы ввели 4 независимых регистра MIR в схему на рис. 4.23. В начале каждого цикла (на рис. 4.2 это время Δw) значение MIR3 копируется в регистр MIR4, значение MIR2 — в регистр

MIR3, значение MIR1 — в регистр MIR2, а в MIR1 загружается новая микрооперация из очереди. Затем каждый регистр MIR выдает сигналы управления, но используются только некоторые из них. Поля A и B из регистра MIR1 применяются для выбора регистров, которые запускают защелки A и B, а поле АЛУ в регистре MIR1 не используется и не связано ни с чем на тракте данных.

В следующем цикле микрооперация передается в регистр MIR2; выбранные регистры находятся в защелках A и B. Поле АЛУ теперь используется для запуска АЛУ. В следующем цикле поле C запишет результаты обратно в регистры. После этого микрооперация передается в регистр MIR4 и инициирует любую необходимую операцию памяти, используя загруженное значение регистра MAR (или MDR для записи).

Нужно обсудить еще один аспект микроархитектуры Mic-4 — микропереходы. Некоторым IJVM-командам нужен условный переход, который осуществляется с помощью бита N. Когда происходит такой переход, конвейер не может продолжать работу. Именно поэтому нам пришлось добавить в микрооперацию бит перехода. Когда в блок формирования очереди поступает микрооперация с таким битом, блок воздерживается от передачи сигнала о получении данных блоку декодирования. В результате машина простаивает до тех пор, пока этот переход не разрешится.

Предположительно, некоторые IJVM-команды, не зависящие от этого перехода, могут быть уже переданы в блок декодирования, но не в блок формирования очереди, поскольку он еще не выдал сигнал о получении. Чтобы разобраться в этой путанице и вернуться к нормальной работе, требуется специальное устройство и особые механизмы, но мы не будем рассматривать их в этой книге. Здесь отметим только, что Эдгар Дейкстра, написавший знаменитый манифест о губительности команд `goto`, был безусловно прав [Dijkstra, 1968a].

Мы начали с микроархитектуры Mic-1 и, пройдя довольно долгий путь, закончили микроархитектурой Mic-4. Аппаратное обеспечение микроархитектуры Mic-1 оказалось очень простым, поскольку практически все управление было реализовано программно. Микроархитектура Mic-4 является конвейеризированной структурой с семью ступенями и более сложным аппаратным обеспечением. Данный конвейер изображен на рис. 4.24. Цифры в кружочках соответствуют компонентам на рис. 4.23. В микроархитектуре Mic-4 поток байтов заранее вызывается из памяти в автоматическом режиме, декодируется в IJVM-команды, которые затем с помощью ПЗУ превращаются в последовательность операций и применяются по назначению. Первые три ступени конвейера при желании можно связать с задающим генератором тракта данных, но работа будет происходить не в каждом цикле. Например, блок выборки команд совершенно точно не сможет передавать новый код операции блоку декодирования в каждом цикле, поскольку выполнение IJVM-команды занимает несколько циклов, и очередь быстро переполнится.

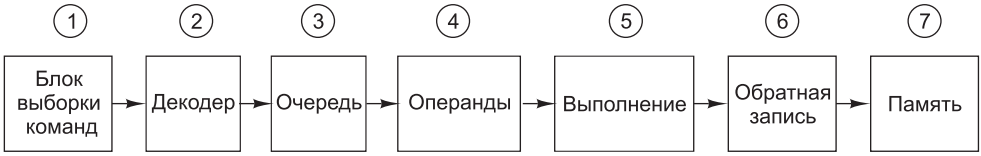


Рис. 4.24. Конвейер Mic-4

В каждом цикле значения регистров MIR смещаются, и микрооперация, находящаяся в начале очереди, копируется в регистр MIR1. Затем сигналы управления от всех четырех регистров MIR передаются по тракту данных, вызывая определенные действия. Каждый регистр MIR контролирует отдельную часть тракта данных и, следовательно, разные микрошаги.

В данной архитектуре используется конвейерный процессор, благодаря чему отдельные шаги становятся очень короткими, а тактовая частота — высокой. Многие процессоры проектируются именно таким образом, особенно те, которым приходится выполнять устаревший набор команд (CISC). Например, реализация Core i7 в некоторых аспектах сходна с микроархитектурой Mic-4, как мы увидим позднее в этой главе.

Повышение производительности

Все производители компьютеров хотят, чтобы их системы работали как можно быстрее. В этом разделе мы рассмотрим ряд передовых технологий повышения производительности системы (в первую очередь — процессора и памяти), которые исследуются в настоящее время. Поскольку в компьютерной индустрии конкуренция очень острая, между появлением новой идеи о повышении скорости работы компьютера и воплощением этой идеи обычно проходит очень немного времени. Следовательно, большинство идей, которые мы сейчас будем обсуждать, вероятнее всего, уже применяются в производстве.

Рассматриваемые усовершенствования можно разделить на две категории, касающиеся реализации и архитектуры. Усовершенствования реализации — это такие способы построения новых процессора и памяти, после применения которых система работает быстрее, но архитектура при этом не меняется. Изменение реализации без изменения архитектуры означает, что устаревшие программы смогут работать на новой машине, а это очень важно для успешной продажи. Чтобы усовершенствовать реализацию, можно, например, использовать более быстрый задающий генератор, но это — не единственный способ. Отметим, что рост производительности от процессора 80386 к процессорам 80486, Pentium и последующих моделей (таких, как Core i7) достигался совершенствованием реализации, тогда как архитектура в целом оставалась неизменной.

Однако некоторые варианты усовершенствований можно реализовать только путем изменения архитектуры. Иногда, например, нужно добавить новые команды или регистры, причем таким образом, чтобы устаревшие программы могли работать на новых моделях. В этом случае для достижения максимальной производительности программное обеспечение приходится переделывать или, по крайней мере, обрабатывать новым компилятором для использования новых возможностей.

Однако один раз в несколько десятилетий разработчики понимают, что старая архитектура уже никуда не годится и единственный способ развивать технологии дальше — начать все заново. Таким революционным скачком было появление в 80-х годах RISC-архитектуры, а сейчас уже приближается следующий прорыв. Мы рассмотрим соответствующий пример (Intel IA-64) в главе 5.

Далее в этом разделе мы расскажем о четырех приемах повышения производительности процессора. Начнем мы с трех хорошо зарекомендовавших себя

вариантов усовершенствования реализации, а затем перейдем к варианту, требующему незначительного усовершенствования архитектуры. Указанные приемы касаются кэш-памяти, прогнозирования переходов, исполнения с изменением очередности и подменой регистров, а также спекулятивного исполнения.

Кэш-память

Одним из самых важных аспектов проектирования компьютеров была и остается организация системы памяти, поддерживающей передачу операндов процессору с той же скоростью, с которой он их обрабатывает. Стремительное увеличение быстродействия процессора, к сожалению, не сопровождается столь же стремительным повышением скорости работы памяти. Относительно процессора память работает все медленнее и медленнее. Учитывая чрезвычайную важность основной памяти, эта ситуация, которая ухудшается с каждым годом, серьезно тормозит развитие высокопроизводительных систем, заставляя разработчиков искать обходные пути.

Современные процессоры предъявляют определенные требования к системе памяти и в плане времени ожидания (задержки в доставке операнда), и в плане пропускной способности (объему данных, передаваемых в единицу времени). К сожалению, эти два аспекта системы памяти в значительной степени противоречивы. Обычно с повышением пропускной способности увеличивается время ожидания. Например, конвейерные технологии, которые мы использовали в микроархитектуре Mic-3, можно применить и к системе памяти, в этом случае запросы к памяти будут обрабатываться более рационально, с перекрытием. Однако, к сожалению, как и в микроархитектуре Mic-3, это приводит к увеличению времени ожидания отдельных операций памяти. С увеличением скорости работы задающего генератора становится все сложнее поддерживать такую систему памяти, которая могла бы передавать операнды за один или два цикла.

Один из вариантов решения проблемы — добавление кэш-памяти. Как мы отмечали в подразделе «Кэш-память» раздела «Основная память» главы 2, в кэш-памяти хранятся наиболее часто используемые слова, за счет чего повышается скорость доступа к ним. Если достаточно большой процент нужных слов находится в кэш-памяти, время ожидания может значительно сократиться.

Одной из самых эффективных технологий одновременного увеличения пропускной способности и уменьшения времени ожидания является применение нескольких блоков кэш-памяти. Основной прием — введение отдельных кэшей для команд и данных (так называемая **разделенная кэш-память**). Такая кэш-память имеет несколько преимуществ, в частности операции могут начинаться независимо в каждой кэш-памяти, что удваивает пропускную способность системы памяти. Именно по этой причине в микроархитектуре Mic-1 нам понадобились два отдельных порта памяти: отдельный порт для каждого кэша. Отметим, что каждый кэш имеет независимый доступ к основной памяти.

В наше время многие системы памяти гораздо сложнее этих. Между разделенной кэш-памятью и основной памятью часто помещается **кэш-память второго уровня**. Вообще говоря, с повышением требований к памяти количество уровней кэш-памяти может достигать трех и более. На рис. 4.25 изображена система с тремя уровнями кэш-памяти. Прямо на микросхеме центрального процессора

находится небольшой кэш для команд (L1-I) и небольшой кэш для данных (L1-D) объемом обычно от 16 до 64 Кбайт. Есть еще кэш-память второго уровня (L2), которая расположена не на самой микросхеме процессора, а рядом с ним в том же блоке. Кэш-память второго уровня соединяется с процессором через высокоскоростной тракт данных. Эта кэш-память обычно не является разделенной и объединяет данные и команды. Ее размер — от 512 Кбайт до 1 Мбайт. Кэш-память третьего уровня (L3) находится на той же плате, что и процессор, и обычно состоит из статического ОЗУ в несколько мегабайтов, которое функционирует гораздо быстрее, чем динамическое ОЗУ основной памяти. Как правило, все содержимое кэш-памяти первого уровня находится в кэш-памяти второго уровня, а все содержимое кэш-памяти второго уровня — в кэш-памяти третьего уровня.

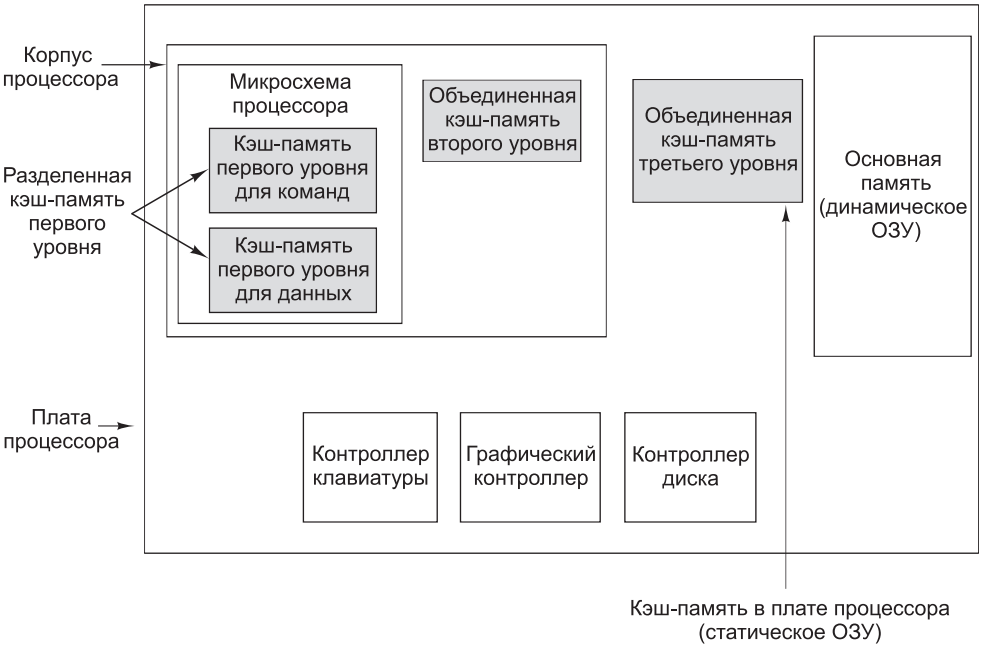


Рис. 4.25. Система с тремя уровнями кэш-памяти

Существует два варианта локализации адресов, от которых зависит работа кэш-памяти. **Пространственная локализация** основана на вероятности того, что в скором времени появится потребность обратиться к ячейкам памяти, расположенным рядом с недавно вызванными ячейками. Исходя из этого наблюдения, в кэш-память переносится больше данных, чем требуется в данный момент. **Временна́я локализация** имеет место, когда недавно вызванные ячейки запрашиваются снова. Это может происходить, например, с ячейками памяти, находящимися рядом с вершиной стека, или с командами внутри цикла. Принцип временно́й локализации используется при выборе элементов, которые следует удалить из кэш-памяти в случае кэш-промаха. Обычно удаляются те элементы, к которым давно не было обращений.

Во всех типах кэш-памяти используется следующая модель. Основная память разделяется на блоки фиксированного размера, которые называются **строками**

кэша. Строка кэша состоит из нескольких последовательных байтов (обычно от 4 до 64). Строки нумеруются, начиная с 0, то есть если размер строки составляет 32 байта, то строка 0 — это байты с 0 по 31, строка 1 — байты с 32 по 63 и т. д. В любой момент в кэш-памяти находится несколько строк. Когда происходит обращение к памяти, контроллер кэш-памяти проверяет, есть ли нужное слово в кэш-памяти. Если слово есть (случай кэш-попадания), то можно сэкономить время, требуемое на доступ к основной памяти. Если данного слова в кэш-памяти нет (случай кэш-промаха), то одна из строк из кэша удаляется, а вместо нее туда помещается запрошенная строка из основной памяти или из кэш-памяти более низкого уровня. Существуют множество вариаций данной схемы, но в их основе всегда лежит идея держать в кэш-памяти как можно больше часто используемых строк, чтобы число кэш-попаданий было максимальным.

Кэш-память прямого отображения

Самый простой тип кэш-памяти — **кэш-память прямого отображения**. Пример одноуровневой кэш-памяти прямого отображения показан на рис. 4.26, *а*. Данная кэш-память содержит 2048 элементов. Каждый элемент (ряд) может вмещать ровно одну сроку из основной памяти. Если размер строки кэша составляет 32 байта (как в этом примере), кэш-память может вмещать 64 Кбайт. Каждый элемент кэш-памяти состоит из трех частей:

- Бит достоверности указывает, есть достоверные данные в элементе или нет. Когда система загружается, все элементы маркируются как недостоверные.

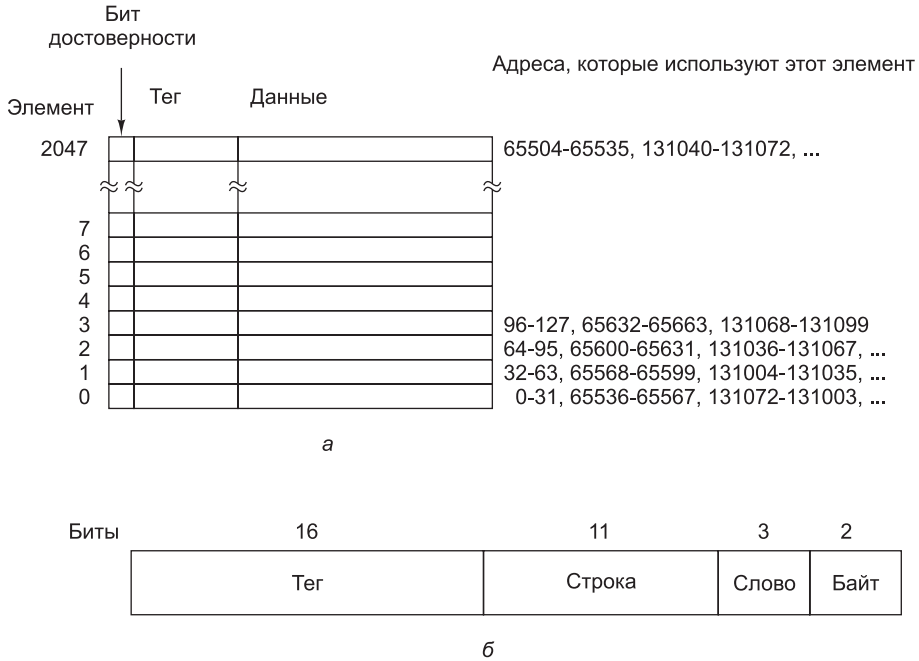


Рис. 4.26. Кэш-память прямого отображения (*а*); 32-разрядный виртуальный адрес (*б*)

- Поле тега состоит из уникального 16-разрядного значения, указывающего соответствующую строку памяти, из которой поступили данные.
- Поле данных содержит копию данных памяти. Это поле вмещает одну строку кэша размером 32 байта.

В кэш-памяти прямого отображения заданное слово может храниться только в одном месте. Если его в этом месте нет, значит, его вообще нет в кэш-памяти. Для хранения данных в кэше и извлечения их из кэша адрес разбивается на 4 компонента, как показано на рис. 4.26, *б*:

- Поле тега соответствует битам, сохраненным в поле тега элемента кэш-памяти.
- Поле строки указывает, какой элемент кэш-памяти содержит соответствующие данные, если они есть в кэш-памяти.
- Поле слова указывает, на какое слово в строке производится ссылка.
- Поле байта обычно не используется, но если требуется только один байт, в этом поле указано, какой именно байт в слове нужен. Для кэш-памяти, поддерживающей только 32-разрядные слова, это поле всегда содержит 0.

Когда центральный процессор выдает адрес памяти, аппаратура выделяет из этого адреса 11 бит поля строки и использует их для поиска в кэш-памяти одного из 2048 элементов. Если элемент действителен, то производится сравнение поля тега основной памяти и поля тега кэш-памяти. Если поля равны, значит, в кэш-памяти есть запрашиваемое слово. Такая ситуация называется **кэш-попаданием**. В случае кэш-попадания слово берется прямо из кэш-памяти и тогда не нужно обращаться к основной памяти. Из элемента кэш-памяти берется только нужное слово, остальная часть элемента не используется. Если элемент кэш-памяти недействителен (недостоверен) или поля тега не совпадают, то нужного слова нет в памяти. Такая ситуация называется **кэш-промахом**. В этом случае 32-байтная строка вызывается из основной памяти и сохраняется в кэш-памяти, заменяя тот элемент, который там был. Однако если существующий элемент кэш-памяти изменяется, его нужно записать обратно в основную память до того, как он будет заменен.

Несмотря на усложнение решения, доступ к нужному слову может быть чрезвычайно быстрым. Поскольку известен адрес, известно и точное местоположение слова, *если оно имеется в кэш-памяти*. Это значит, что нужно считать слово из кэш-памяти, доставить его процессору и одновременно с этим проверить, правильное ли это слово (путем сравнения полей тега). Поэтому процессор в действительности получает слово из кэш-памяти одновременно или даже до того, как становится известно, запрошенное это слово или нет.

При такой схеме смежные строки основной памяти помещаются в смежные элементы кэш-памяти. Фактически, в кэш-памяти может храниться до 64 Кбайт смежных данных. Однако две строки, адреса которых отличаются ровно на 64 Кбайт (65 536 байт) или на любое целое, кратное от этого числа, не могут одновременно храниться в кэш-памяти (поскольку они имеют одно и то же значение поля строки). Например, если программа обращается к данным с адресом X, а затем выполняет команду, которой требуются данные с адресом $X + 65\,536$ (или с любым другим адресом в той же строке), вторая команда требует перезагрузки элемента кэш-памяти. Если это происходит достаточно часто, то могут возник-

нуть проблемы. Если кэш-память плохо работает, лучше, чтобы ее вообще не было, поскольку при каждой операции с основной памятью считывается целая строка, а не одно слово.

Кэш-память прямого отображения — это самый распространенный тип кэш-памяти, и она достаточно эффективна, поскольку коллизии, подобные описанной, случаются крайне редко или вообще не случаются[1]. Например, качественный компилятор может учитывать подобные коллизии при размещении команд и данных в памяти. Отметим, что указанный случай не произойдет в системе, где команды и данные хранятся раздельно, поскольку конфликтующие запросы будут обслуживаться разными кэшами. Таким образом, мы видим второе преимущество наличия двух кэшей вместо одного — больше гибкости при разрешении конфликтных ситуаций.

Ассоциативная кэш-память с множественным доступом

Как было отмечено ранее, различные строки основной памяти конкурируют за право занять одну и ту же область кэша. Если программе, использующей кэш-память, изображенную на рис. 4.26, *а*, часто требуются слова с адресами 0 и 65 536, то будут иметь место постоянные конфликты, поскольку каждое обращение потенциально повлечет за собой вытеснение из кэш-памяти той или иной строки. Чтобы разрешить эту проблему, нужно сделать так, чтобы в каждом элементе кэш-памяти помещалось по две и более строк. Кэш-память с n возможными элементами для каждого адреса называется **n-входовой ассоциативной кэш-памятью**. 4-входовая ассоциативная кэш-память изображена на рис. 4.27.

Рис. 4.27. 4-входовая ассоциативная кэш-память

Ассоциативная кэш-память с множественным доступом по сути гораздо сложнее, чем кэш-память прямого отображения, поскольку хотя элемент кэш-памяти и можно вычислить по адресу основной памяти, требуется проверить n

[1] На самом деле подобные коллизии не столь уж и редки из-за того, что при страничном способе организации виртуальной памяти и параллельном исполнении нескольких заданий страницы как бы «перемешиваются». Разбиение программы на страницы осуществляется случайным образом, поэтому и «локальность кода» может быть нарушена. — *Примеч. науч. ред.*

элементов кэш-памяти, чтобы узнать, есть ли там нужная нам строка. При этом проверка должна выполняться очень быстро. Тем не менее практика показывает, что 2- или 4-входовая ассоциативная кэш-память дает хороший результат, поэтому внедрение этих дополнительных схем вполне оправдано.

Использование ассоциативной кэш-памяти с множественным доступом ставит разработчика перед выбором. Если нужно поместить новый элемент в кэш-память, какой именно из старых элементов удалить? Для большинства задач хорошо подходит алгоритм обработки элемента, который **дольше всего не использовался** (Least Recenly Used, **LRU**). Имеется определенный порядок каждого набора ячеек, доступных из данной ячейки памяти. Всякий раз, когда осуществляется доступ к любой строке, в соответствии с алгоритмом LRU список обновляется, и маркируется элемент, к которому произведено последнее обращение. Когда требуется заменить какой-нибудь элемент, удаляется тот, который находится в конце списка, то есть тот, который использовался раньше других.

Возможен также предельный случай — 2048-входовая ассоциативная кэш-память, содержащая единственный набор из 2048 элементов. В данном случае все адреса памяти оказываются в этом наборе, поэтому при поиске требуется сравнивать нужный адрес со всеми 2048 тегами в кэш-памяти. Отметим, что для этого каждый элемент кэш-памяти должен содержать специальную логическую схему. Поскольку поле строки в данном случае нулевое, поле тега — это весь адрес за исключением полей слова и байта. Более того, когда строка кэша заменяется, возможными кандидатами на смену являются все 2048 элементов. Хранение упорядоченного списка из 2048 элементов потребовало бы громоздкого управления системными ресурсами, поэтому применение алгоритма LRU оказывается неэффективным. (Вспомните, что список должен обновляться при каждой операции с памятью.) Интересно, что кэш-память с большим числом входов далеко не всегда превосходит по производительности кэш-память, в которой число входов невелико, а в некоторых случаях работает даже хуже. Поэтому число входов больше четырех встречается редко.

Наконец, особой проблемой для кэш-памяти является запись. Когда процессор записывает слово, а это слово есть в кэш-памяти, он, очевидно должен либо обновить слово, либо удалить данный элемент кэш-памяти. Практически во всех разработках имеет место обновление кэш-памяти. А что же можно сказать об обновлении копии в основной памяти? Эту операцию можно отложить на потом до того момента, когда строка кэша будет готова к замене в соответствие с алгоритмом LRU. Выбор труден и ни одно из решений не является предпочтительным. Немедленное обновление элемента основной памяти называется **сквозной записью**. Этот подход обычно гораздо проще реализуется и, к тому же, он более надежен, поскольку современная память при ошибке всегда может восстановить свое предыдущее состояние. К сожалению, при этом требуется передавать больше данных в память, поэтому в сложных проектах стремятся использовать альтернативный подход — **обратную**, или **отложенную**, **запись**.

С процессом записи связана еще одна проблема: что происходит, если нужно записать что-либо в ячейку, которой нет в кэш-памяти? Должны ли данные передаваться в кэш или просто записываться в основную память? И снова ни один из ответов не является во всех отношениях лучшим. В большинстве разработок,

в которых применяется обратная запись, данные передаются в кэш-память. Эта технология называется **заполнением по записи** (write allocation). С другой стороны, в тех разработках, где применяется сквозная запись, элемент в кэш-память при записи обычно не помещается, поскольку это усложняет систему. Заполнение по записи полезно только в том случае, если имеют место повторные записи в одно и то же слово или в разные слова в пределах одной строки кэша.

Эффективность кэширования является крайне важным условием повышения общей производительности системы в силу огромного разрыва между быстродействием процессора и памяти. Дискуссия об альтернативных стратегиях кэширования ведется постоянно [Sanchez and Kozyrakis, 2011; Gaur et. al, 2011].

Прогнозирование переходов

Современные компьютеры в значительной степени конвейеризированы. Конвейер, изображенный на рис. 4.24, имеет семь ступеней; более сложно организованные компьютеры содержат конвейеры с десятью и более ступенями. Конвейеризация лучше работает с линейным кодом, поэтому блок выборки команд может просто считывать последовательные слова из памяти и отправлять их в блок декодирования заранее, еще до того, как они понадобятся.

Единственная проблема состоит в том, что эта модель совершенно нереалистична. Программы вовсе не являются линейными последовательностями команд — в них полно команд переходов. Рассмотрим простые команды листинга 4.4. Переменная `i` сравнивается с 0 (вероятно, на практике это самое распространенное сравнение). В зависимости от результата другой переменной, `k`, присваивается одно из двух возможных значений.

Листинг 4.4. Фрагмент программы

```
if(i==0)
        k=1;
else
        k=2;
```

Возможный вариант трансляции на язык ассемблера показан в листинге 4.5. Язык ассемблера мы будем рассматривать позже в этой книге, а сейчас достаточно знать, что программа, более или менее похожая на программу из листинга 4.5, вполне возможна. Первая команда сравнивает переменную `i` с нулем. Вторая совершает переход к метке `Else` (начало секции `else`), если `i` не равно 0. Третья команда присваивает значение 1 переменной `k`. Четвертая команда выполняет переход к следующему оператору программы. Компилятор поместил там метку `Next`. Пятая команда присваивает значение 2 переменной `k`.

Листинг 4.5. Программа из листинга 4.4 после трансляции на язык ассемблера

```
        CMP i, 0    ; сравнение i с 0
        BNE Else    ; переход к Else, если они не равны
Then:   MOV k, 1    ; присваивание значения 1 переменной k
        BR Next     ; безусловный переход к Next
Else:   MOV k, 2    ; присваивание значения 2 переменной k
Next:
```

Мы видим, что две из пяти команд являются командами перехода. Более того, одна из них, `BNE`, — это команда условного перехода (переход, который

осуществляется тогда и только тогда, когда выполняется определенное условие, в данном случае — это неравенство двух операндов предыдущей команды `CMP`). Самый длинный линейный код состоит здесь из двух команд, поэтому очень трудно организовать высокоскоростной конвейер.

На первый взгляд может показаться, что безусловные переходы, например команда `BR Next` в листинге 4.5, не влекут за собой никаких проблем. Вообще говоря, в данном случае нет никакой двусмысленности в том, куда дальше идти. Почему же блок выборки команд не может просто продолжать считывать команды с целевого адреса (то есть с того места, куда осуществляется переход)?

Сложность объясняется самой природой конвейера. На рис. 4.24, например, мы видим, что декодирование происходит на второй ступени. Следовательно, блоку выборки команд приходится решать, откуда вызывать следующую команду еще до того, как он узнает, команда какого типа только что была получена. Только в очередном цикле он сможет выяснить, что получил команду безусловного перехода, хотя еще до этого он вызывает команду, следующую за командой безусловного перехода. То есть в значительной части конвейеризированных машин (например, UltraSPARC III) сначала выполняется команда, *следующая после* команды безусловного перехода, хотя по логике вещей так быть не должно. Позиция после перехода называется **слотом отсрочки** (delay slot). Core i7 (а также машина, используемая в листинге 4.5) не поддерживают слот отсрочки, а обойти эту проблему путем внутреннего усложнения часто сложно. Оптимизирующий компилятор постарается найти какую-нибудь полезную команду, чтобы поместить ее в слот отсрочки, но часто ничего подходящего нет, поэтому компилятор вынужден вставлять туда команду `NOP`. Хотя программа остается корректной, объем ее растет и работает она медленнее.

С условными переходами дело обстоит еще хуже. Во-первых, они тоже содержат слоты отсрочки, во-вторых, блок выборки команд узнает, откуда нужно считывать команду, гораздо позже. Первые конвейеризированные машины просто **простаивали**, пока выяснялось, нужно совершать переход или нет. Простой по три или четыре цикла при каждом условном переходе, особенно если 20 % команд являются командами условного перехода, значительно снижает производительность.

Поэтому большинство машин прогнозируют, будет выполнен встретившийся условный переход или нет. Для этого, например, можно предполагать, что все условные переходы назад будут выполняться, а все условные переходы вперед нет. Что касается первой части предположения, то команды перехода назад обычно помещаются в конце циклов, а большинство циклов выполняются многократно, поэтому предположение о переходе к началу цикла чаще всего будет правильным.

Со второй частью предположения дело обстоит сложнее. Некоторые переходы вперед осуществляются в случае обнаружения ошибки в программе (например, невозможность открытия файла). Ошибки случаются редко, поэтому в большинстве случаев подобные переходы не происходят. Естественно, существуют множество переходов вперед, никак не связанных с ошибками, поэтому процент успеха здесь не так высок, как в случае перехода назад. Однако это все же лучше, чем ничего.

Если переход спрогнозирован правильно, то ничего особенного делать не нужно. Просто продолжается выполнение программы. Проблема возникает тогда, когда переход спрогнозирован неправильно. Вычислить, куда нужно перейти,

и перейти именно туда, несложно. Самое сложное — отменить уже выполненные команды, которые не нужно было выполнять.

Существует два способа отмены команд. Первый способ — продолжать выполнять команды, вызванные после спрогнозированного условного перехода до тех пор, пока одна из этих команд не попытается изменить состояние машины (например, сохранить значение в регистре). Тогда вместо того, чтобы перезаписывать регистр, нужно поместить вычисленное значение во временный (скрытый) регистр, а затем, когда выяснится, что прогноз был правильным, просто скопировать это значение в обычный регистр. Второй способ — сохранять (например, в скрытом временном регистре) значение любого регистра, который может быть переписан. В результате машина сможет вернуться в предыдущее состояние в случае неправильно спрогнозированного перехода. Реализация обоих подходов очень сложна и требует громоздкой системы учета использования системных ресурсов. А если встретится второй условный переход еще до того, как станет известно, был ли правильно спрогнозирован первый условный переход, ситуация может совершенно запутаться.

Динамическое прогнозирование переходов

Ясно, что точные прогнозы очень ценны, поскольку позволяют процессору работать с полной скоростью. В настоящее время проводится множество исследований, целью которых является усовершенствование алгоритмов прогнозирования переходов [Chen et al., 2003; Falcon et al., 2004; Jimenez, 2003; Parikh et al., 2004]. Один из подходов — хранить (в особом устройстве) специальную таблицу, в которую центральный процессор будет записывать условные переходы, когда они встретятся. Если условный переход встретится снова, его можно будет найти в этой таблице. Простейшая версия такой схемы показана на рис. 4.28, *а*. В данном случае таблица содержит по одной ячейке для каждой команды условного перехода. В ячейке находится адрес команды перехода, а также бит, который указывает, произошел ли переход, когда эта команда встретилась в последний раз. Прогноз заключается в выборе того же пути, по которому программа пошла в предыдущий раз при выполнении команды перехода. Если прогноз оказывается неправильным, бит в таблице меняется.

Существует несколько вариантов организации данной таблицы. Фактически они полностью аналогичны вариантам организации кэш-памяти. Рассмотрим машину с 32-разрядными командами, которые расположены таким образом, что два младших бита каждого адреса памяти равны 00. Таблица содержит 2^n ячеек (строк). Из команды перехода можно извлечь $n + 2$ младших бита и осуществить сдвиг вправо на два бита. Это n-разрядное число можно использовать в качестве индекса в таблице, проверяя, совпадает ли адрес, сохраненный там, с адресом перехода. Как и в случае с кэш-памятью, здесь нет необходимости сохранять $n + 2$ младших бита, поэтому их можно опустить (то есть сохраняются только старшие адресные биты — тег). Если адреса совпали, бит прогнозирования используется для прогнозирования перехода. Если тег неправильный или элемент недействителен, значит, имеет место промах. В этом случае можно применять правило перехода вперед/назад.

Если таблица динамики переходов содержит, скажем, 4096 элементов, то адреса 0, 16384, 32768 и т. д. будут конфликтовать; аналогичная проблема

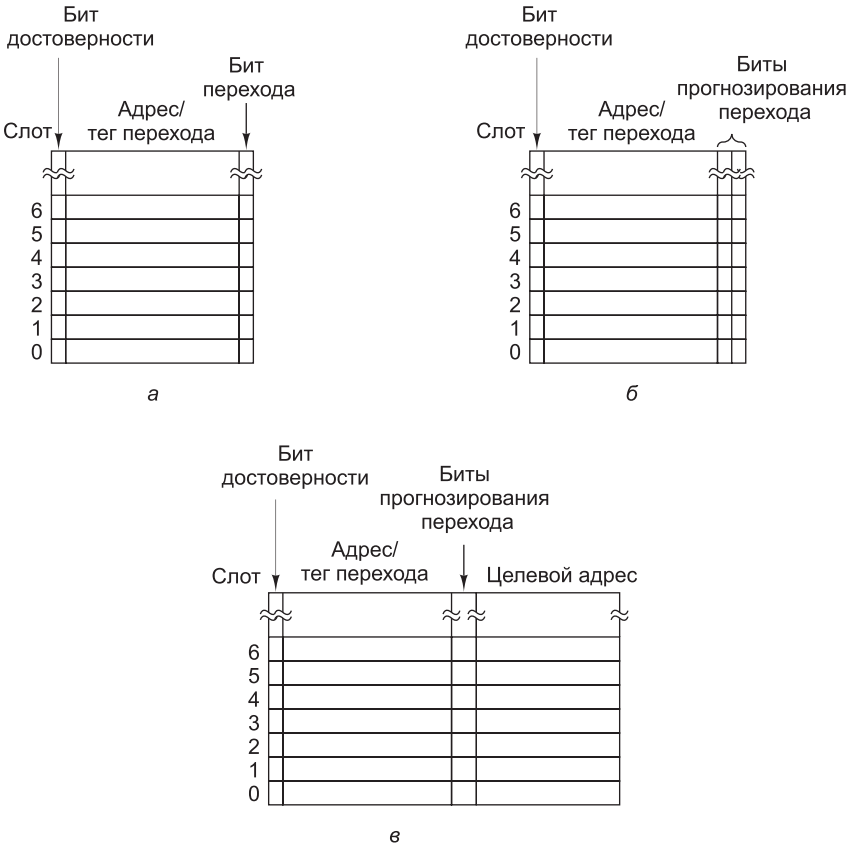


Рис. 4.28. Таблица динамики переходов с 1-разрядным указателем перехода (*а*); таблица динамики переходов с 2-разрядным указателем перехода (*б*); соответствие между адресом команды перехода и целевым адресом (*в*)

встречается и при кэшировании. Здесь возможно такое же решение: 2-альтернативный, 4-альтернативный или n-альтернативный ассоциативный элемент. Как и в случае кэш-памяти, предельный случай — один n-альтернативный ассоциативный элемент.

При достаточно большом размере таблицы и достаточной степени ассоциативности эта схема хорошо подходит для большинства ситуаций. Тем не менее одна систематическая проблема возникает всегда. При выходе из цикла переход предсказывается неправильно, и, что еще хуже, этот неправильный прогноз изменяет бит в таблице, который после этого будет показывать, что переход совершать не требуется. То есть в следующий раз, когда опять потребуется выполнять цикл, переход в конце первого прохода цикла окажется спрогнозированным неправильно. Если цикл находится внутри другого цикла или внутри часто вызываемой процедуры, эта ошибка будет повторяться достаточно часто.

Для решения проблемы можно немного изменить метод прогнозирования, чтобы прогноз менялся не после одного, а только после двух последовательных неправильных прогнозов. Такой подход требует наличия в таблице двух битов прогнозиро-

вания переходов (рис. 4.28, *б*): один должен указывать, предполагается совершать переход или нет, а второй — был сделан переход в прошлый раз или нет.

Этот алгоритм можно представить в виде конечного автомата с четырьмя состояниями (рис. 4.29). После ряда последовательных успешных прогнозов отсутствия перехода конечный автомат будет находиться в состоянии 00 и в следующий раз также покажет, что перехода нет. Если этот прогноз окажется неправильным, автомат перейдет в состояние 01, но в следующий раз он все равно покажет отсутствие перехода. Только в том случае, если это последний прогноз тоже окажется ошибочным, конечный автомат перейдет в состояние 11, прогнозируя наличие перехода. Фактически левый бит — это прогноз, а правый бит — то, что случилось в прошлый раз (то есть был ли переход). В данной разработке используются только 2 бита, но возможно применение и 4, и 8 бит.

Рис. 4.29. 2-разрядный конечный автомат для прогнозирования переходов

Это не первый конечный автомат, который мы рассматриваем. На рис. 4.19 тоже изображен конечный автомат. На самом деле все наши микропрограммы можно считать конечными автоматами, поскольку каждая строка представляет особое состояние, в котором может находиться автомат, с четко определенными переходами к конечному набору других состояний. Конечные автоматы очень широко используются при разработке аппаратного обеспечения.

До сих пор мы предполагали, что цель каждого условного перехода известна. Обычно либо в явном виде давался адрес, к которому нужно перейти (он содержался непосредственно в команде), либо было известно смещение относительно текущей команды (то есть число со знаком, которое нужно было прибавить к счетчику команд). Часто это предположение имеет силу, но некоторые команды условного перехода вычисляют целевой адрес, предварительно выполняя определенные арифметические действия над значениями регистров. Даже если взять конечный автомат, изображенный на рис. 4.29, который точно прогнозирует переходы, прогноз окажется невостребованным, поскольку неизвестен целевой адрес. Один из возможных выходов из подобной ситуации — сохранить в таблице адрес, к которому был осуществлен переход в прошлый раз, как показано на рис. 4.28, *в*. Тогда, если в таблице указано, что в прошлый раз, когда встретилась

команда перехода по адресу 516, переход был совершен к адресу 4000, то целевым снова будет адрес 4000 (в случае, если предсказывается переход).

Еще один подход к прогнозированию перехода — следить, были ли совершены последние k условных переходов независимо от того, какие это были команды. Это k-разрядное число, которое хранится в **сдвиговом регистре динамики переходов**, затем сравнивается параллельно со всеми элементами таблицы с k-разрядным ключом, и в случае совпадения применяется тот прогноз, который найден в этом элементе. Удивительно, но эта технология работает достаточно хорошо.

Статическое прогнозирование переходов

Все технологии прогнозирования переходов, которые обсуждались до сих пор, являются динамическими, то есть выполняются во время работы программы. Они приспосабливаются к текущему режиму работы программы, и это их положительное качество. Отрицательной стороной этих технологий является то, что они требуют специализированной и дорогостоящей аппаратуры, и микросхемы для реализации этих технологий получаются очень сложными.

Можно пойти другим путем, призвав на помощь компилятор. Обратите внимание на следующий оператор:

```
for (i = 0; i < 1000000; i++) { ... }
```

Когда компилятор встречает такой оператор, он знает, что переход в конце цикла будет происходить практически всегда. Если бы был способ сообщить это аппаратуре, можно было бы избавиться от огромного объема работы.

Хотя такой подход связан с изменением архитектуры (а не только реализации), в некоторых машинах, например UltraSPARC III, помимо обычных команд условного перехода (которые нужны для обратной совместимости), имеется еще один набор команд. Новые команды содержат бит, по которому компилятор определяет, совершать переход или не совершать. Когда встречается такой бит, блок выборки команд просто делает то, что ему положено. Более того, нет необходимости тратить драгоценное пространство в таблице динамики переходов для этих команд, что сокращает количество конфликтных ситуаций.

Наконец, наша последняя технология прогнозирования переходов основана на профилировании [Fisher and Freudenberger, 1992]. Это тоже статическая технология, только в данном случае программа не заставляет компилятор вычислять, какие переходы нужно совершать, а какие нет. Программа реально выполняется, а переходы фиксируются; эта информация поступает в компилятор, который затем использует специальные команды условного перехода для того, чтобы сообщить аппаратуре, что нужно делать.

Исполнение с изменением последовательности и подмена регистров

Большинство современных процессоров являются и конвейеризированными и суперскалярными, как показано на рис. 2.5. Это означает, что имеется блок выборки команд (IFU), который заранее вызывает команды из памяти и передает их в блок декодирования. Блок декодирования, в свою очередь, передает декодированные команды соответствующим функциональным блокам для выполнения.

В некоторых случаях блок декодирования может разбивать отдельные команды на микрооперации перед тем, как отправить их функциональным блокам.

Ясно, что проще всего выполнять все команды в том порядке, в котором они вызываются из памяти (предполагается, что прогнозирование переходов всегда оказывается верным). Однако из-за взаимозависимости команд такое последовательное выполнение не всегда дает оптимальную производительность. Если команде требуется значение, которое вычисляется предыдущей командой, вторая команда не сможет выполняться, пока первая не выдаст нужную величину. Таким образом, в ситуации реальной взаимозависимости второй команде приходится ждать. Существуют и другие виды взаимозависимостей, но о них мы поговорим позже.

Чтобы обойти эти проблемы и достичь оптимальной производительности, некоторые процессоры пропускают взаимозависимые команды и переходят к следующим (независимым) командам. Естественно, что при этом алгоритм распределения команд должен давать такой же результат, как если бы все команды выполнялись в том порядке, в котором они написаны. А теперь продемонстрируем на конкретном примере, как происходит переупорядочение команд.

Чтобы продемонстрировать суть проблемы, начнем с машины, которая запускает команды в том порядке, в котором они расположены в программе, и требует, чтобы выполнение команд также завершалось в порядке, соответствующем программному. Важность второго требования прояснится позднее.

Наша машина содержит 8 доступных программисту регистров, от R0 до R7. Все арифметические команды используют три регистра: два для операндов и один для результата, как и в микроархитектуре Mic-4. Мы предполагаем, что если команда декодируется в цикле n, выполнение начинается в цикле $n + 1$. В случае простой команды, например команды сложения или вычитания, запись обратно в выходной регистр происходит в конце цикла $n + 2$. В случае с более сложной командой, например командой умножения, запись в регистр происходит в конце цикла $n + 3$. Чтобы сделать наш пример реалистичным, мы позволим блоку декодирования выдавать до двух команд за цикл. Коммерческие суперскалярные процессоры могут генерировать 4 или даже 6 команд за цикл.

Последовательность выполнения команд иллюстрирует табл. 4.11. В первом столбце приводится номер цикла, во втором — номер команды, в третьем — сама команда. В четвертом столбце показаны выданные команды (максимум две команды за цикл). Цифры в пятом столбце сообщают, какие команды завершены. Помните, что в нашем примере мы требуем, чтобы команды и запускались, и завершались в строго определенном порядке, поэтому выдача команды $k + 1$ может произойти только после выдачи команды k, а результат команды $k + 1$ не может быть записан в выходной регистр до того, как завершится команда k. Оставшиеся 16 столбцов мы обсудим позже.

После декодирования команды блок декодирования должен определить, запускать команду сразу или нет. Для этого блок декодирования должен знать состояние всех регистров. Если, например, текущей команде требуется регистр, значение которого еще не посчитано, текущая команда не выдается, и центральный процессор вынужден простаивать.

Следить за состоянием регистров призвано специальное устройство — **счетчик обращений** (scoreboard), впервые появившийся в системе CDC 6600. Для каждого регистра счетчик обращений содержит небольшую схему, которая

подсчитывает, сколько раз этот регистр используется выполняющимися командами в качестве источника. Если одновременно может выполняться максимум 15 команд, будет достаточно 4-разрядного счетчика. Когда запускается команда, элементы счетчика обращений, соответствующие регистрам операндов, увеличиваются на 1. Когда выполнение команды завершено, соответствующие элементы счетчика уменьшаются на 1.

Счетчик обращений содержит аналогичные схемы подсчета для целевых регистров. Поскольку допускается только одна запись за раз, эти схемы могут быть размером в один бит. Правые 16 столбцов в табл. 4.11 демонстрируют показания счетчика обращений.

В реальных машинах счетчик обращений также следит за использованием функционального блока, чтобы избежать выдачи команды, для которой нет доступного функционального блока. Для простоты мы предполагаем, что подходящий функциональный блок всегда есть в наличии, поэтому функциональные блоки в таблице не показаны.

В первой строке табл. 4.11 представлена команда 1, которая перемножает значения регистров R0 и R1 и помещает результат в регистр R3. Поскольку ни один из этих регистров еще не используется, команда запускается, а счетчик обращений показывает, что регистры R0 и R1 считываются, а регистр R3 записывается. Ни одна из последующих команд не может записывать результат в эти регистры и не может считывать регистр R3 до тех пор, пока не завершится выполнение команды 1. Поскольку это команда умножения, она закончится в конце цикла 4. Значения счетчика обращений, приведенные в каждой строке, отражают состояние регистров после запуска команды, записанной в этой же строке. Пустые клетки соответствуют значению 0.

Поскольку в качестве примера рассматривается суперскалярная машина, которая может запускать две команды за цикл, вторая команда выдается также во время цикла 1. Она складывает значения регистров R0 и R2, а результат сохраняет в регистре R4. Чтобы определить, можно ли запускать эту команду, применяются следующие правила:

1. Если какой-нибудь операнд записывается, запускать команду нельзя (RAW-взаимозависимость).
2. Если считывается регистр результатов, запускать команду нельзя (WAR-взаимозависимость).
3. Если записывается регистр результатов, запускать команду нельзя (WAW-взаимозависимость).

Мы уже рассматривали реальные взаимозависимости (RAW-взаимозависимости), имеющие место, когда команде в качестве источника нужно использовать результат предыдущей команды, которая еще не завершилась. Два других типа взаимозависимостей менее серьезные. По существу, они связаны с конфликтами ресурсов. При **WAR-взаимозависимости** (Write After Read — **запись после чтения**) одна команда пытается перезаписать регистр, который предыдущая команда еще не закончила считывать. **WAW-взаимозависимость** (Write After Write — **запись после записи**) похожа на WAR-взаимозависимость. Подобной взаимозависимости можно избежать, если вторая команда будет помещать результат где-либо в другом месте еще (возможно, временно). Если ни одна из трех

Таблица 4.11. Суперскалярный процессор с последовательной выдачей и последовательным завершением команд

					Считываемые регистры								Записываемые регистры							
Цикл	**#**	**Команда**	**Выдача**	**Завершение**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1 2	R3 = R0 × R1 R4 = R0 + R2	1 2		1 2	1 1	 1									1 1	 1			
2	3 4	R5 = R0 + R1 R6 = R1 + R4	3 –		3 3	2 2	1 1									1 1	1 1	1 1		
3					3	2	1									1	1	1		
4				1 2 3	2 1	1 1	1										1	1 1		
5	 5	 R7 = R1 × R2	4 5			1 2	 1		1 1										1 1	 1
6	6	R1 = R0 – R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9	 7	 R3 = R3 × R1	6 –		1 1		1 1							1 1						
10					1		1							1						
11				6																
12	 8	 R1 = R4 + R4	7 –			1 1		1 1								1 1				
13						1		1								1				
14						1		1								1				
15				7																
16			8						2					1						
17									2					1						
18				8																

упомянутых ситуаций не возникает и нужный функциональный блок доступен, команду можно выдать. В этом случае команда 2 использует регистр R0, который в данный момент считывается незаконченной командой, но подобное перекрытие допустимо, поэтому команда 2 может запускаться. Сходным образом команда 3 запускается во время цикла 2.

А теперь перейдем к команде 4, которая должна использовать регистр R4. К сожалению, из таблицы мы видим, что в регистр R4 в данный момент производится запись (см. строку 3 в таблице). Здесь имеет место RAW-взаимозависимость, поэтому блок декодирования простаивает до тех пор, пока регистр R4 не станет доступным. Во время простоя блок декодирования прекращает получать команды из блока выборки команд. Когда внутренние буферы блока выборки команд заполнятся, он прекращает вызывать команды из памяти.

Следует упомянуть, что следующая команда, команда 5, не конфликтует ни с одной из завершенных команд. Ее можно было бы декодировать и выдать, если бы в нашей системе не требовалось, чтобы команды выдавались строго по порядку.

Посмотрим, что происходит в цикле 3. Команда 2, а это команда сложения (два цикла), завершается в конце цикла 3. Но ее результат не может быть сохранен в регистре R4 (который тогда освободится для команды 4). Почему? Из-за необходимости записи результатов в регистры в соответствии с порядком выполнения программы. Но зачем? Что плохого произойдет, если сохранить результат в регистре R4 сейчас и сделать это значение доступным?

Ответ на этот вопрос очень важен. Предположим, что команды могут завершаться в произвольном порядке. Тогда в случае прерывания будет очень сложно сохранить состояние машины так, чтобы его можно было потом восстановить. В частности, нельзя будет сказать, что все команды до какого-то адреса были выполнены, а все команды после этого адреса выполнены не были, как было бы при так называемом **точном прерывании**, которое является желательной характеристикой центрального процессора [Moudgill and Vassiliadis, 1996]. Сохранение результатов в произвольном порядке делает прерывания неточными, и именно поэтому в некоторых машинах требуется соблюдение жесткого порядка завершения команд.

Вернемся к нашему примеру. В конце цикла 4 результаты всех трех команд могут быть сохранены, поэтому в цикле 5 может быть выдана команда 4, а также недавно декодированная команда 5. Всякий раз, когда завершается какая-нибудь команда, блок декодирования должен проверять, нет ли простаивающей команды, которую уже можно выдать.

В цикле 6 команда 6 простаивает, потому что ей нужно записать результат в регистр R1, а регистр R1 занят. Выполнение команды начинается только в цикле 9. Чтобы завершить всю последовательность из 8 команд, требуется 18 циклов из-за многочисленных ситуаций взаимозависимости, хотя аппаратура способна выдавать по две команды за цикл. По колонкам «Выдача» и «Завершение» табл. 4.11 видно, что все команды выдаются из блока декодирования по порядку и завершаются эти команды тоже по порядку.

Рассмотрим альтернативный подход: исполнение с изменением последовательности. В такой системе выполнение команд может начинаться в произвольном порядке и завершаться также в произвольном порядке. В табл. 4.12 показана та же последовательность из восьми команд, только теперь разрешен произвольный порядок выдачи команд и сохранения результатов в регистрах.

Таблица 4.12. Работа суперскалярного процессора с изменением очередности запуска и завершения команд

					Считываемые регистры								Записываемые регистры							
Цикл	**#**	**Команда**	**Выдача**	**Завершение**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1 2	R3 = R0 × R1 R4 = R0 + R2	1 2		1 2	1 1	 1									1 1	 1			
2	3 4	R5 = R0 + R1 R6 = R1 + R4	3 –		3 3	2 2	1 1									1 1	1 1	1 1		
3	5 6	R7 = R1 × R2 S1 = R0 – R2	5 6	2	3 4 3	3 3 3	2 3 2									1 1 1	1 1	1 1 1		1 1 1
4	7 8	R3 = R3 × S1 S2 = R4 + R4	4 – 8	1 3	3 3 3 2 1	4 4 4 3 2	2 2 2 2 2		1 1 3 3 3							1 1 1		1 1 1 1	1 1 1 1 1	1 1 1 1 1
5				6		2	1		3					1					1	1
6			7	4 5 8		2 1	1 1	1 1 1 1	3 2 2					1 1 1		1 1 1 1			1	1 1
7								1								1				
8								1								1				
9				7																

Первое отличие встречается в цикле 3. Несмотря на то что команда 4 простаивает, мы можем декодировать и запустить команду 5, поскольку она не создает конфликтной ситуации ни с одной из выполняющихся команд. Однако пропуск команд порождает новую проблему. Предположим, что команда 5 использует операнд, который вычисляется пропущенной командой 4. При текущем состоянии счетчика обращений мы этого не заметим. В результате нам придется расширить счетчик обращений, чтобы следить за записями, которые совершают пропущенные команды. Это можно сделать, добавив еще одно битовое отображение, по одному биту на регистр, для контроля за записями, которые делают простаивающие команды (эти счетчики в таблице не показаны). Правило запуска команд следует расширить, с тем чтобы предотвратить запуск команды, операнд которой должен был быть записан предшествующей, но пропущенной командой.

Теперь посмотрим на команды 6–8 в табл. 4.11. Здесь мы видим, что команда 6 помещает вычисленное значение в регистр R1 и это значение используется командой 7. Мы также видим, что это значение больше не требуется, потому что команда 8 переписывает значение регистра R1. Нет никакой надобности использовать регистр R1 для хранения результата команды 6. Еще хуже то, что далеко не лучшим является выбор регистра R1 в качестве промежуточного, хотя с точки зрения программиста, привыкшего к идее последовательного выполнения команд без перекрытий, этот выбор является самым разумным.

В табл. 4.12 мы ввели новый метод для решения этой проблемы — **подмену регистров** (register renaming). Блок декодирования заменяет регистр R1 в командах 6 (цикл 3) и 7 (цикл 4) скрытым для программиста регистром S1. После этого команда 6 может запускаться одновременно с командой 5. Современные процессоры содержат десятки скрытых регистров, которые используются для подмены. Такая технология часто позволяет устранить WAR- и WAW-взаимозависимости.

В команде 8 мы снова применяем подмену регистров. На этот раз регистр R1 заменяется регистром S2, поэтому операция сложения может начаться до того, как освободится регистр R1, а освободится он только в конце цикла 6. Если окажется, что результат в этот момент должен быть в регистре R1, содержимое регистра S2 всегда можно скопировать туда. Еще лучше то, что все будущие команды, использующие этот результат, смогут в качестве источника задействовать регистр подмены, в котором действительно хранится нужное значение. В любом случае, выполнение команды 8 начнется раньше.

В настоящих (не гипотетических) компьютерах подмена регистров происходит с многократным вложением. Существуют множество скрытых регистров и таблица, в которой показывается соответствие доступных программисту и скрытых регистров. Например, чтобы найти местоположение регистра R0, нужно обратиться к элементу 0 этой таблицы. На самом деле реального регистра R0 нет, а есть только связь между именем R0 и одним из скрытых регистров. Эта связь часто меняется во время выполнения программы, чтобы избежать взаимозависимостей.

Обратите внимание на четвертый и пятый столбец табл. 4.12. Вы видите, что команды запускаются не по порядку и завершаются также не по порядку. Вывод весьма прост: изменяя последовательность выполнения команд и подменяя регистры, мы можем ускорить процесс вычислений почти в два раза.

Спекулятивное исполнение

В предыдущем разделе мы ввели понятие переупорядочения команд, необходимого для повышения производительности. В действительности имелось в виду переупорядочение команд в пределах одного базового блока программы. Рассмотрим этот аспект подробнее.

Компьютерные программы можно разбить на **базовые блоки**, каждый из которых представляет собой линейную последовательность команд с точкой входа в начале и точкой выхода в конце. Базовый блок не содержит никаких управляющих структур (например, условных операторов `if` или операторов цикла `while`), поэтому при трансляции на машинный язык никакие переходы не создаются. Базовые блоки связываются операторами управления.

Программа в такой форме может быть представлена в виде ориентированного графа, как показано на рис. 4.30. Здесь мы вычисляем сумму кубов четных и нечетных целых чисел до какого-либо предела и помещаем результаты в переменные `evensum` и `oddsum` соответственно (листинг 4.6). В пределах каждого базового блока технологии, упомянутые в предыдущем подразделе, работают отлично.

Листинг 4.6. Фрагмент программы

```
evensum=0;
oddsum=0;
i=0;
while (i<limit) {
        k=i*i*i;
        if(((i/2)*2)==i)
            evensum=evensum+k;
        else
            oddsum=oddsum+k;
        i=i+1;
}
```

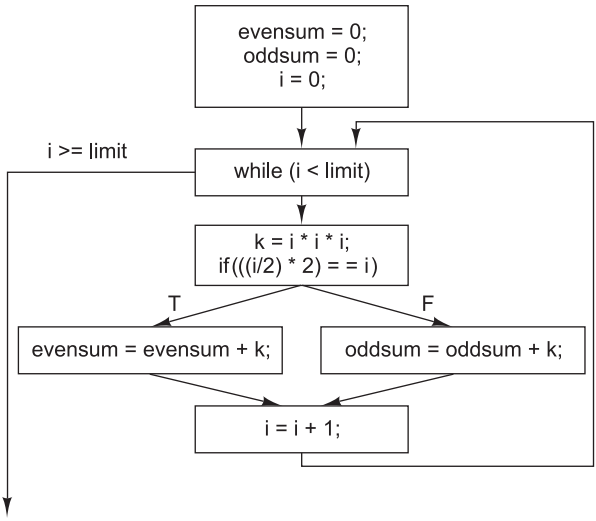


Рис. 4.30. Граф базового блока для фрагмента программы, приведенного в листинге 4.6

Проблема состоит в том, что большинство базовых блоков очень короткие, что не позволяет обеспечить достаточную степень параллелизма при их выполнении. Следовательно, нужно сделать так, чтобы механизм переупорядочения последовательности команд можно было применять не только в пределах конкретного базового блока. Полезнее всего будет передвинуть потенциально медленную операцию в графе повыше, чтобы ее выполнение началось раньше. Это может быть команда `LOAD`, операция с плавающей точкой или даже начало длинной цепочки зависимостей. Перемещение кода вверх по ребру графа называется **подъемом**.

Посмотрите еще раз на рис. 4.30. Представим, что все переменные, кроме `evensum` и `oddsum`, были помещены в регистры. Тогда имело бы смысл переместить команды `LOAD` в начало цикла до вычисления переменной `k`, чтобы выполнение этих команд началось раньше, а полученные результаты были доступны в момент, когда они понадобятся. Естественно, при каждой итерации требуется только одно значение, поэтому остальные команды `LOAD` будут отбрасываться, но если кэш-память и основная память конвейеризированы, то подобная процедура имеет смысл. Выполнение команды раньше того, как станет известно, понадобится эта команда или нет, называется **спекулятивным исполнением**. Чтобы использовать эту технологию, требуется поддержка компилятора, аппаратного обеспечения, а также некоторое усовершенствование архитектуры. В большинстве случаев переупорядочение команд за пределами одного базового блока находится вне возможностей аппаратного обеспечения, поэтому компилятор должен перемещать команды явным образом.

Спекулятивное исполнение команд создает некоторые интересные проблемы. Например, очень важно, чтобы ни одна из спекулятивных команд не давала результата, который невозможно отменить, поскольку позднее может оказаться, что эту команду не нужно было выполнять. Обратимся к листингу 4.6 и рис. 4.30. Очень удобно производить сложение, как только появляется значение `k` (даже до условного оператора `if`), но при этом нежелательно сохранять результаты в памяти. Чтобы предотвратить перезапись регистров до того, как станет известно, нужны ли полученные результаты, нужно подменить все выходные регистры, которые используются спекулятивной командой. Как вы можете себе представить, счетчик обращений для отслеживания всех этих ситуаций очень сложен, но при наличии соответствующего аппаратного обеспечения его вполне можно создать.

Однако при наличии спекулятивных команд возникает еще одна проблема, которую нельзя решить путем подмены регистров. Что произойдет, если спекулятивная команда вызовет исключение? В качестве примера можно привести команду `LOAD`, которая вызывает кэш-промах в компьютере со строкой кэша достаточно большого размера (скажем, 256 байт) и памятью, которая работает гораздо медленнее, чем центральный процессор и кэш. Если нам требуется команда `LOAD` и работа машины останавливается на несколько циклов, пока загружается строка кэша, то это не так страшно, поскольку данное слово действительно нужно. Но если машина простаивает для вызова слова, которое, как окажется позднее, нам ни к чему, это совершенно нерационально. Если подобных «оптимизаций» будет слишком много, то центральный процессор будет работать медленнее, чем если бы «оптимизаций» вообще не было. (Если машина содержит виртуальную память, о которой рассказывается в главе 6, то спекулятивное выполнение ко-

манды LOAD может даже вызвать обращение к отсутствующей странице. Подобные ошибки могут значительно повлиять на производительность, поэтому важно их избегать.)

В ряде современных компьютеров данная проблема решается следующим образом. В них поддерживается специальная команда SPECULATIVE-LOAD, которая производит попытку вызвать слово из кэш-памяти, а если слова там нет, просто прекращает вызов. Если значение в кэше обнаруживается и оно действительно требуется, его можно использовать, а если его в кэше нет, аппаратное обеспечение должно сразу же его получить. Если затем окажется, что данное значение нам не нужно, то никаких потерь времени не будет.

Более сложную ситуацию можно проиллюстрировать следующим оператором:

```
if (x > 0) z = y/x;
```

Здесь x, y и z — переменные с плавающей точкой. Предположим, что все эти переменные поступают в регистры заранее, а команда деления с плавающей точкой (эта команда выполняется медленно) перемещается вверх по графу и выполняется еще до условного оператора if. К сожалению, если значение x равно 0, то программа завершается в результате попытки деления на 0. Таким образом, спекулятивная команда приводит к сбою в изначально правильной программе. Еще хуже то, что программист изменяет программу, чтобы предотвратить подобную ситуацию, но сбой все равно происходит.

Одно из возможных решений — специальные версии тех команд, которые могут вызывать исключения. Кроме того, к каждому регистру добавляется так называемый **бит отравления** (poison bit). Если спекулятивная команда дает сбой, она не инициирует перехват исключения, а устанавливает бит отравления в регистр результатов. Если затем этот регистр используется обычной командой, выполняется перехват исключения (как и должно быть в случае исключения). Однако если этот результат не используется, бит отравления сбрасывается и никак не влияет на ход выполнения программы.

Примеры уровня микроархитектуры

В этом разделе в свете материала, изучаемого в этой главе, мы рассмотрим три современных процессора. Наше изложение будет кратким, поскольку компьютеры чрезвычайно сложны, содержат миллионы вентилей, и у нас нет возможности давать подробное описание. Процессоры, предлагаемые в качестве примеров, те же, что и раньше — Core i7, OMAP4430 и ATmega168.

Микроархитектура процессора Core i7

На первый взгляд, Core i7 кажется вполне традиционной CISC-машиной с большим и громоздким набором команд, поддерживающим 8-, 16- и 32-разрядные целочисленные операции, а также 32- и 64-разрядные операции с плавающей точкой. В нем всего 8 доступных регистров на процессор, причем ни один из них не повторяет другие. Допустимая длина команд составляет 1–17 байт. В общем, налицо стандартная унаследованная архитектура, которая все делает не так.

На самом же деле, процессор Core i7 основан на современном эффективном RISC-ядре с развитой конвейеризацией. Его тактовая частота уже очень высока, а в последующие годы, скорее всего, вырастет еще больше. Удивительно, как инженерам Intel на основе архаичной архитектуры удалось построить процессор, отвечающий всем современным требованиям. Итак, в этом подразделе мы рассмотрим микроархитектуру Core i7 и разберемся в принципах ее работы.

Обзор микроархитектуры Sandy Bridge

Микроархитектура Core i7, называемая **Sandy Bridge**, ознаменовала собой решительный отход от микроархитектур предыдущих поколений, включая P4 и P6. Примерная схема микроархитектуры Core i7 изображена на рис. 4.31.

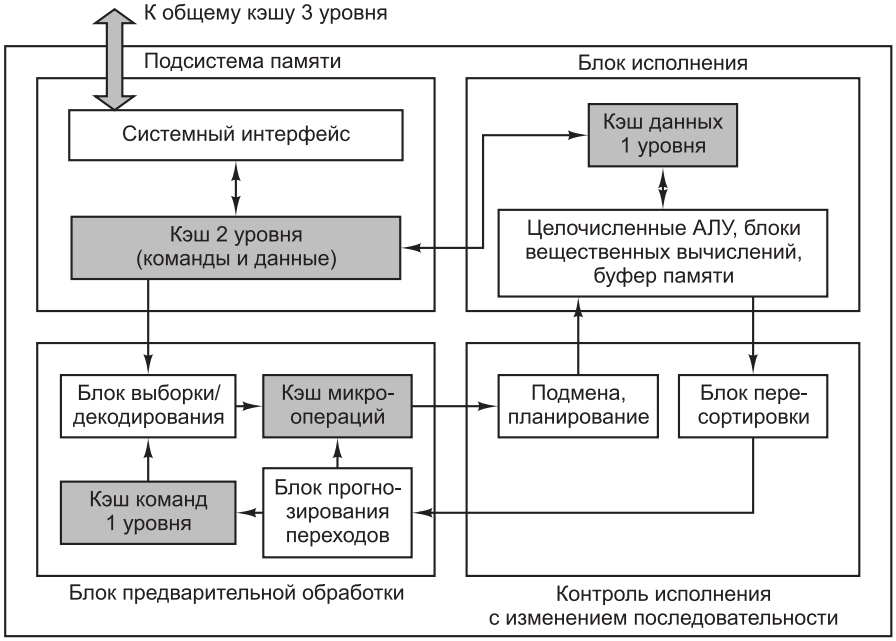


Рис. 4.31. Микроархитектура Core i7

Core i7 состоит из четырех основных блоков: подсистемы памяти, блока предварительной обработки, блока контроля исполнения с изменением последовательности и блока исполнения. Рассмотрим эти блоки по порядку, начиная с верхнего левого и продвигаясь против часовой стрелки.

Каждый процессор Core i7 содержит подсистему памяти с объединенным кэшем второго уровня (L2), а также логикой доступа к кэшу 3 уровня (L3). Все процессоры совместно используют общий кэш 3 уровня — это «последняя остановка», после которой обращение выходит за пределы микросхемы центрального процессора и отправляется по шине в очень длинное путешествие к внешней памяти. Объем кэшей L2 в Core i7 составляет 256 Кбайт; они представляют собой 8-входовую ассоциативную кэш-память с 64-байтовыми строками. Размер общего кэша L3 лежит в диапазоне от 1 до 20 Мбайт. Чем больше вы заплатите Intel, тем больше будет размер кэша. Независимо от размера кэш L3 представляет собой

12-входовый ассоциативный кэш с 64-байтовыми строками. Если запрос к кэшу третьего уровня не приносит результата, он передается во внешнюю память по шине DDR3.

С кэшем первого уровня связаны два блока предварительной выборки (они не показаны на рисунке), которые пытаются перенести данные из основной памяти в L1 еще до того, как эти данные запрошены. Один блок осуществляет предварительную выборку следующего блока памяти при обнаружении последовательного «потока» памяти, передаваемого процессору. Второй, более сложный блок предварительной выборки отслеживает последовательность адресов операций чтения/записи конкретной программы. Если операции осуществляются с постоянным шагом (например, 0x1000...0x1020...0x1040...), блок заранее выбирает следующий элемент, к которому, скорее всего, обратится программа. Предварительная выборка с постоянным шагом чрезвычайно эффективно работает в программах, перебирающих массивы структурированных переменных.

Подсистема памяти на рис. 4.31 связана как с блоком предварительной обработки, так и с кэшем данных L1. Блок предварительной обработки отвечает за выборку команд из подсистемы памяти, декодирование их в микрооперации «в стиле RISC» и сохранение в двух кэшах команд. Все команды после выборки помещаются в кэш команд L1 (первого уровня). Размер кэша L1 составляет 32 Кбайт, он представляет собой 8-входовую ассоциативную кэш-память с 64-байтовыми строками. В ходе выборки из кэша L1 команды попадают в декодеры, определяющие последовательность микроопераций, используемых для реализации команды в конвейере исполнения.

Механизм декодирования позволяет «навести мосты» между устаревшим набором CISC-команд и современным трактом данных RISC.

Декодированные микрооперации передаются в кэш микроопераций, который Intel называет «кэшем команд L0 (нулевого уровня)». Кэш микроопераций напоминает традиционный кэш команд, но в нем достаточно места для хранения последовательностей микрокоманд, генерируемых отдельными командами. Поскольку кэшируются не исходные команды, а декодированные микрооперации, необходимость в повторном декодировании при последующих исполнениях команды отпадает. На первый взгляд может показаться, что фирма Intel сделала это для ускорения работы конвейера (и в самом деле, кэширование ускоряет процесс генерирования команды), но Intel утверждает, что кэш микроопераций был добавлен для сокращения энергопотребления блоком предварительной обработки. При наличии кэша микроопераций остальная часть блока 80 % времени проводит в режиме «сна» с минимальным энергопотреблением.

Прогнозирование переходов также выполняется в блоке предварительной обработки. Блок прогнозирования должен «угадать», когда ход выполнения программы отклонится от строго последовательной выборки, причем он должен сделать это задолго до исполнения команд перехода. В Core i7 этот блок работает весьма эффективно. К сожалению, в большинстве архитектур подробная информация о блоках прогнозирования переходов хранится в секрете. Дело в том, что производительность блока прогнозирования часто является важнейшим фактором, определяющим общую скорость архитектуры. Чем больше точности прогнозирования выжмут проектировщики из каждого квадратного миллиметра кремния, тем выше производительность всей архитектуры. По этой

причине компании держат свои секреты «под замком» и даже угрожают своим работникам судебным преследованием, если те захотят поделиться драгоценными знаниями. Достаточно сказать, что все блоки прогнозирования отслеживают результаты предыдущих переходов и используют эту информацию для новых прогнозов. Секрет составляют подробный перечень отслеживамых показателей, способ их хранения и использования. В конце концов, если вы вдруг изобретете какой-нибудь невероятный способ предсказания будущего, вы тоже не станете выкладывать его в Интернет на всеобщее обозрение.

Команды передаются из кэша микроопераций планировщику команд в порядке, определяемом программой, но при их исполнении возможно отступление от этого порядка. Обнаружив микрооперацию, которую нельзя исполнить, планировщик команд удерживает ее, одновременно продолжая обрабатывать поток команд — запускаются все последующие команды, которые не требуют обращения к занятым ресурсам (регистрам, функциональным блокам и т. д.). Здесь же выполняется подмена регистров, благодаря чему WAR- и WAW-взаимозависимые команды могут исполняться без задержки.

Хотя очередность выдачи команд может отличаться от предусмотренной в программе, требование точности прерываний архитектуры Core i7 гласит, что результаты выполнения ISA-команд должны становиться видимыми программе без отступления от заданной программой последовательности. За реализацию этого требования отвечает блок пересортировки.

Блоки исполнения непосредственно осуществляют целочисленные операции, операции с плавающей точкой и специализированные команды. Существуют несколько блоков исполнения, работающих параллельно. Данные они получают из регистрового файла и кэша данных первого уровня.

Конвейер Sandy Bridge у Core i7

На рис. 4.32 приведена упрощенная схема микроархитектуры Sandy Bridge, в том числе ее конвейер. В верхней части схемы находится блок предварительной обработки, ответственный за выборку команд из памяти и их подготовку к исполнению. Этот блок получает новые команды x86 из кэша команд первого уровня. Они декодируются в микрооперации и помещаются в кэш микроопераций, содержащий приблизительно 1,5K микроопераций. По своей производительности он сопоставим с традиционным кэшем нулевого уровня на 6 Кбайт. В кэше микроопераций каждые шесть микроопераций объединяются в группу, занимающую одну строку. Для формирования более протяженных последовательностей микроопераций применяется объединение строк.

Если блок декодирования сталкивается с условным переходом, он обращается за информацией к блоку прогнозирования переходов. Этот блок содержит историю переходов, осуществлявшихся в прошлом, и на основании накопленных данных предполагает, будет ли выполнен условный переход, когда он в следующий раз встретится в программе. Именно здесь используются секретные алгоритмы, о которых говорилось выше.

Если команда перехода отсутствует в таблице, применяется статическое прогнозирование. При этом подразумевается, что обратный переход, во-первых, является частью цикла, во-вторых, по умолчанию предполагается, что он будет выполнен. Точность статического прогноза в этом случае очень высока. Прямой

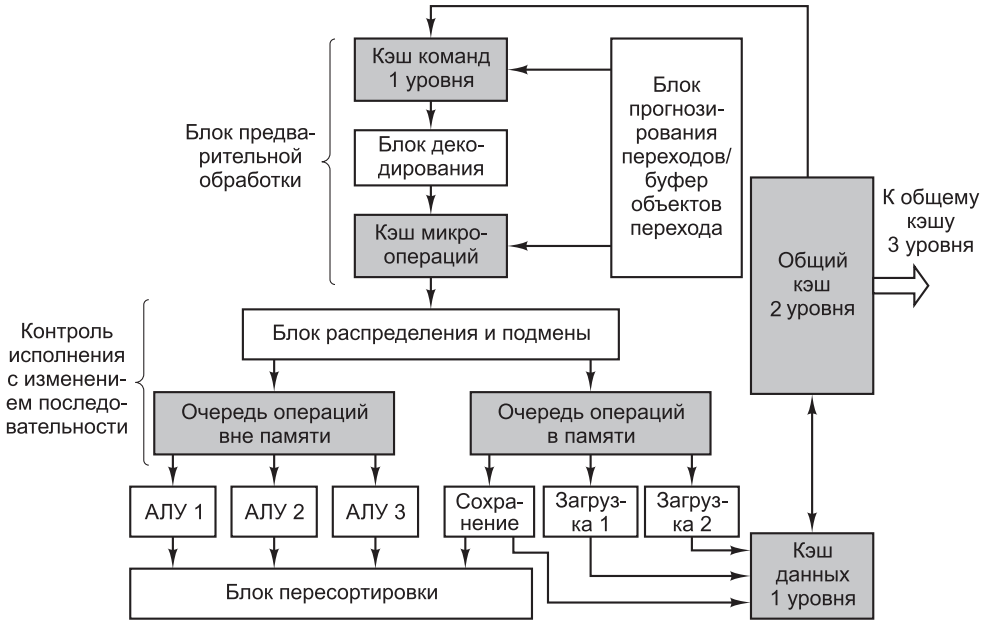


Рис. 4.32. Упрощенная схема тракта данных Core i7

переход считается входящим в структуру оператора `if` и не выполняемым по умолчанию. Точность статического прогноза в случае прямых переходов значительно ниже, чем в случае обратных.

Для выбранной ветви целевой адрес определяется по содержимому **буфера объектов перехода**, или **BTB**. В BTB хранится целевой адрес перехода при последнем выполнении. Обычно этот адрес правилен (собственно, он всегда правилен для переходов с постоянным смещением). Косвенные переходы (например, используемые при вызовах виртуальных функций и в командах C++ `switch`) осуществляются по разным адресам и их прогнозирование по данным BTB будет ошибочным.

Второй компонент конвейера — логика исполнения с изменением последовательности — получает данные из кэша микроопераций. При поступлении из блока предварительной обработки каждой последующей микрооперации (а за цикл их поступает три) **блок распределения и подмены** регистрирует ее в таблице, состоящей из 168 записей и называемой **буфером переупорядочивания команд** (ReOrder Buffer, **ROB**). В этом буфере хранятся данные о состоянии микрооперации, вплоть до пересортировки ее результатов. Затем блок распределения и подмены проводит проверку на предмет доступности ресурсов, необходимых для выполнения микрооперации. Если ресурсы свободны, микрооперация устанавливается в одну из очередей **планировщика**. Для микроопераций, исполняемых в памяти и вне памяти, предусмотрены отдельные очереди. Если исполнение микрооперации в данный момент невозможно, она откладывается, однако обработка последующих микроопераций продолжается; таким образом, микрооперации часто исполняются вне их исходной последовательности. Этот принцип позволяет поддерживать загрузку всех функциональных блоков на

максимально высоком уровне. В каждый отдельно взятый момент могут одновременно обрабатываться до 154 команд, причем 64 из них могут загружаться из памяти, а 36 — сохраняться в памяти.

Иногда микрооперации простаивают. Это происходит в тех случаях, когда к одному и тому же регистру для чтения или записи пытаются обратиться несколько микроопераций; соответственно, одной из них это удается, а остальным — нет. Такие конфликты, как мы уже выяснили, называются WAR- и WAW-взаимозависимостями. Подмена целевого регистра позволяет записать результаты исполнения микрооперации в один из 160 временных регистров, а значит, выполнить эту микрооперацию немедленно. Если же все временные регистры недоступны или микрооперация попадает в ситуацию RAW-взаимозависимости (обойти которую нельзя), планировщик указывает характер возникшей проблемы в виде записи в буфере ROB. Впоследствии, после освобождения всех необходимых ресурсов, микрооперация устанавливается в одну из очередей на исполнение.

Очереди планировщика помещают готовые к исполнению операции в один из шести функциональных блоков:

1. АЛУ 1 и блок умножения с плавающей точкой.
2. АЛУ 2 и блок сложения/вычитания с плавающей точкой.
3. АЛУ 3, блок обработки переходов и сравнений с плавающей точкой.
4. Команды сохранения.
5. Команды загрузки 1.
6. Команды загрузки 2.

Поскольку планировщики и АЛУ могут обрабатывать по одной операции за цикл, производительность планировщика процессора Core i7 с тактовой частотой 3 ГГц достигает 18 млрд целочисленных операций в секунду; впрочем, на практике такая скорость никогда не достигается. Хотя функциональные блоки не могут обеспечить полную загрузку исполнительных ресурсов, они обладают достаточной исполнительной мощностью, поэтому блок контроля исполнения с изменением последовательности так старательно подыскивает им работу.

Три целочисленных АЛУ не одинаковы. АЛУ 1 выполняет любые арифметические и логические операции, умножения и деления. АЛУ 2 способно выполнять только арифметические и логические операции. АЛУ 3 выполняет арифметические и логические операции, а также разрешение переходов. Не идентичны и два блока исполнения операций с плавающей точкой. Первый поддерживает арифметические операции с плавающей точкой, включая умножение, а второй способен выполнять только сложение и вычитание с плавающей точкой, а также перемещения.

АЛУ и блоки исполнения операций с плавающей точкой получают данные от двух регистровых файлов емкостью по 128 записей. Один из этих файлов отводится для целых чисел, другой — для чисел с плавающей точкой. В них содержатся все операнды, необходимые для исполнения команд; кроме того, они играют роль хранилища результатов. В силу подмены регистров восемь из них содержат регистры, доступные на уровне архитектуры команд (EAX, EBX, ECX, EDX и т. д.), однако расположение «реальных» значений в каждом конкретном случае зависит от изменений в отображении, происходящих в ходе исполнения.

В архитектуре Sandy Bridge появилась технология **AVX** (Advanced Vector Extensions), обеспечивающая поддержку 128-разрядных векторных операций, параллельных по данным (как с целочисленными векторами, так и с векторами с плавающей точкой). В новом расширении ISA размер вектора вдвое увеличился по сравнению с более ранними ISA-расширениями SSE и SSE2. Как архитектура реализует 256-разрядные операции с 128-разрядными трактами данных и функциональными блоками? Два 128-разрядных порта планировщика объединяются для формирования одного 256-разрядного функционального блока.

Кэш данных первого уровня тесно связан с внутренней конвейерной подсистемой Sandy Bridge. В этом 32-килобайтном кэше могут храниться целые числа, числа с плавающей точкой и другие типы данных. В отличие от кэша микроопераций, эти данные никоим образом не декодируются. Функция кэша данных сводится к хранению копий байтов, находящихся в памяти. Что касается его характеристик, то кэш данных первого уровня представляет собой 8-входовую ассоциативную кэш-память с емкостью строки 64 байт. Он поддерживает сквозную запись; иными словами, при изменении строки кэша она незамедлительно копируется обратно в кэш второго уровня. В течение цикла кэш данных первого уровня может выполнить две операции чтения и одну операцию записи. Для реализации множественных обращений используются **банки**, то есть кэш делится на несколько внутренних кэшей (8 в случае Sandy Bridge). Если все три обращения относятся к разным банкам, они могут выполняться одновременно; в противном случае одно из обращений к конфликтующим банкам простаивает. Если затребованное слово не удается обнаружить в кэше первого уровня, отправляется запрос в кэш второго уровня; последний в такой ситуации либо отвечает сразу, либо обращается к общему кэшу третьего уровня, после чего отвечает. В любой момент в состоянии исполнения могут находиться до десяти запросов, направленных из кэша первого уровня в кэш второго уровня.

Так как микрооперации исполняются вне исходной последовательности, сохранение в кэше первого уровня возможно только после пересортировки результатов всех команд, предшествующих команде сохранения. Такую пересортировку результатов с их трассировкой (отслеживанием того, где они находятся) выполняет **блок пересортировки**. В случае прерывания прекращается обработка всех команд, еще не прошедших пересортировку результатов; таким образом, обеспечивается соблюдение требования, согласно которому при прерывании должны быть завершены все команды до определенной точки в программе.

Если команда сохранения прошла пересортировку результатов, но предшествующие команды еще обрабатываются, из-за невозможности обновления кэша первого уровня результаты их исполнения передаются в буфер незавершенных команд. В этом буфере можно единовременно разместить до 36 команд сохранения. Если одна из последующих команд загрузки попытается считать сохраненные данные, она из буфера незавершенных команд будет перенаправлена непосредственно к команде, которая в этот момент еще не помещена в кэш данных первого уровня. Этот процесс называется **перенаправлением для загрузки** (store-to-load forwarding). Хотя механизм перенаправления может показаться элементарным, на практике он реализуется достаточно сложно — может оказаться что незавершенные операции сохранения еще не вычислили свои адреса. В этом

случае микроархитектура не может точно знать, какая из операций сохранения выдаст нужное значение.

Итак, достаточно очевидно, что Core i7 обладает сложной микроархитектурой, проектное решение которой определено необходимостью поддержки унаследованного набора команд Pentium на современном RISC-ядре с высоким уровнем конвейеризации. Эта цель достигается путем деления команд Pentium на микрооперации, их кэширования и передачи (по три микрооперации за раз) конвейеру, где они исполняются с помощью нескольких АЛУ, которые в оптимальных условиях обрабатывают до шести микроопераций за цикл. Микрооперации исполняются с отклонением от исходной последовательности, но возвращаются и сохраняются в кэш-памяти первого и второго уровней в заданном порядке.

Микроархитектура OMAP4430

В однокристальной системе OMAP4430 центральное место занимают два процессора ARM Cortex A9. Cortex A9 — высокопроизводительная микроархитектура, реализующая набор команд ARM (версия 7). Процессор был спроектирован фирмой ARM Ltd; его различные модификации встречаются в широком спектре встроенных систем. ARM не производит процессор, а только поставляет свои разработки производителям электронных микросхем, желающих внедрить его в свои однокристальные системы (в данном случае это фирма Texas Instruments).

Процессор Cortex A9 представляет собой 32-разрядную машину с 32-разрядными регистрами и 32-разрядным трактом данных. Шина памяти, как и внутренняя архитектура, является 32-разрядной. В отличие от Core i7 процессор Cortex A9 изначально проектировался как полноценная RISC-система. Следовательно, необходимости в сложном механизме преобразования старых CISC-команд в микрооперации в данном случае не было. Команды ядра представляют собой готовые микрооперации. Впрочем, за последние годы были добавлены более сложные графические и мультимедийные команды, для исполнения которых требуются специальные устройства.

Обзор микроархитектуры Cortex A9

Структурная схема микроархитектуры Cortex A9 представлена на рис. 4.33. В целом, она значительно проще микроархитектуры Sandy Bridge, применяемой в системах Core i7, что объясняется меньшей сложностью архитектуры системы команд. Тем не менее по некоторым базовым компонентам сходство с Core i7 прослеживается. В первую очередь, это обусловлено технологическими и экономическими факторами. К примеру, в обеих архитектурах применяется многоуровневая иерархия кэширования для удовлетворения жестких ограничений по стоимости, действующих для типичных встроенных приложений; однако размер последнего уровня кэш-памяти Cortex A9 (L2) составляет всего 1 Мбайт; в этом он существенно уступает процессору Core i7, у которого кэш последнего уровня (L3) может достигать 20 Мбайт. Различия связаны по большей части с тем, что во втором случае разработчикам пришлось обеспечить поддержку унаследованного набора CISC-команд, а в первом такой задачи не ставилось.

В верхней части рис. 4.33 изображен 4-входовый ассоциативный кэш команд емкостью 32 Кбайт с 32-байтными строками. Поскольку большинство команд

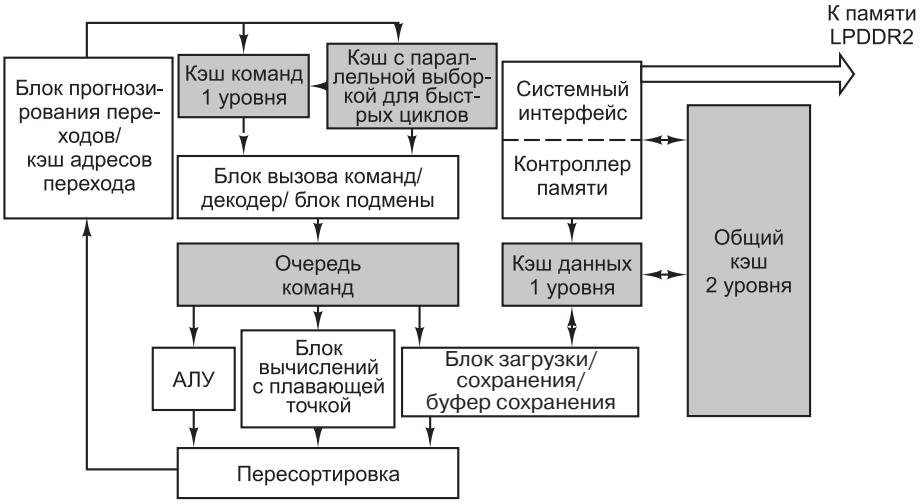


Рис. 4.33. Структурная схема микроархитектуры Cortex A9 однокристальной системы OMAP4430

ARM занимают 4 байта, в этом кэше можно одновременно разместить около 8000 команд — заметно больше, чем у кэша микроопераций Core i7.

Блок вызова команд подготавливает для исполнения до четырех команд за цикл. В случае неудачного обращения в кэш-память первого уровня количество вызываемых команд уменьшается. При обнаружении условного перехода происходит обращение к **кэшу адресов перехода** емкостью 4000 записей; на основе его содержания прогнозируется наличие или отсутствие перехода. Кроме того, если блок предварительной обработки обнаруживает, что программа работает в плотном цикле (небольшой цикл, не являющийся вложенным), она загружает его в специальный кэш (кэш с параллельной выборкой для быстрых циклов). Эта оптимизация ускоряет выборку команд и сокращает энергопотребление, так как кэши и блоки прогнозирования во время выполнения плотного цикла могут находиться в «спящем» режиме.

Результаты работы блока вызова команд передаются декодерам, которые решают, какие ресурсы и входные данные понадобятся командам. Как и в Core i7, после декодирования осуществляется подмена для устранения WAR-конфликтов, способных замедлить исполнение с изменением последовательности. После переименования команды помещаются в очередь распределения команд, которая выдает их при готовности входных данных для функциональных блоков (возможно, с изменением последовательности).

Как видно из рис. 4.33, очередь распределения команд отправляет команды функциональным блокам. Блок целочисленных вычислений содержит два АЛУ и короткий конвейер для команд перехода. Также в нем содержится физический регистровый файл, содержащий регистры ISA, и некоторые временные регистры. Конвейер Cortex A9 также может содержать одно или несколько вычислительных ядер, работающих как дополнительные функциональные блоки. ARM поддерживает вычислительное ядро для вычислений с плавающей точкой **VFP** и векторное целочисленное ядро SIMD, называемое **NEON**.

Блок загрузки/сохранения занимается выполнением различных команд загрузки и сохранения. Он связан с кэшем данных и буфером сохранения. Размер **кэша данных** составляет 32 Кбайт, он представляет собой 4-входовой ассоциативный кэш данных L1 с 32-байтовыми строками. В **буфере сохранения** содержатся команды сохранения, которые еще не записали свои значения в кэш данных (при пересортировке). Выполняемая команда загрузки сначала пытается получить свое значение из буфера выполнения, используя механизм перенаправления для загрузки, аналогичный соответствующему механизму Core i7. Если значение отсутствует в буфере сохранения, оно будет получено из кэша данных. В одном из возможных результатов выполнения команды загрузки буфер сохранения приказывает ожидать, потому что выполнение команды блокируется более ранней операцией сохранения с неизвестным адресом. Если обращение к кэшу данных L1 завершается промахом, блок памяти выбирается из объединенного кэша L2. В некоторых обстоятельствах Cortex A9 также выполняет на аппаратном уровне опережающую выборку данных из кэша L2 в L1, чтобы повысить эффективность операций загрузки и сохранения.

Микросхема OMAP 4430 также содержит логику управления обращениями к памяти. Эта логика разделена на две части: системный интерфейс и контроллер памяти. Системный интерфейс взаимодействует с памятью по 32-разрядной шине LPDDR2. Все внешние запросы к памяти проходят через этот интерфейс. Шина LPDDR2 передает 26-разрядный адрес для 8 банков, возвращающих 32-разрядное слово данных. Теоретически объем основной памяти по каждому каналу LPDDR2 может составлять до 2 Гбайт. В OMAP4430 таких каналов два, поэтому система способна адресовать до 4 Гбайт внешней памяти.

Контроллер памяти преобразует 32-разрядные виртуальные адреса в 32-разрядные физические адреса. Cortex A9 поддерживает виртуальную память (см. главу 6) с размером страницы 4 Кбайт. Для ускорения процесса преобразования предусмотрены специальные таблицы, называемые **буферами быстрого преобразования** (Translation Lookaside Buffers, **TLB**). Они сравнивают текущий виртуальный адрес с адресами, по которым производились обращения в недавнем прошлом.

Конвейер Cortex A9 в OMAP4430

Конвейер Cortex A9 из 11 ступеней; в упрощенной форме он изображен на рис. 4.34. В левой части рисунка ступени обозначены сокращениями. Рассмотрим каждую из них в отдельности. Открывает конвейер ступень *Fe1* (Fetch 1 — выборка 1). В ней адрес следующей команды используется для индексирования кэша команд и начала прогнозирования перехода. Обычно этот адрес следует непосредственно за следующей командой, однако линейная последовательность выполнения может быть нарушена по разным причинам — скажем, если предыдущая команда является командой спрогнозированного перехода, перехвата исключения или прерывания, которое необходимо обработать. Так как выборка команд и прогнозирование переходов занимают более одного цикла, ступень *Fe2* предоставляет дополнительное время для выполнения этих операций. На ступени *Fe3* выбранные команды (до четырех) заносятся в очередь команд.

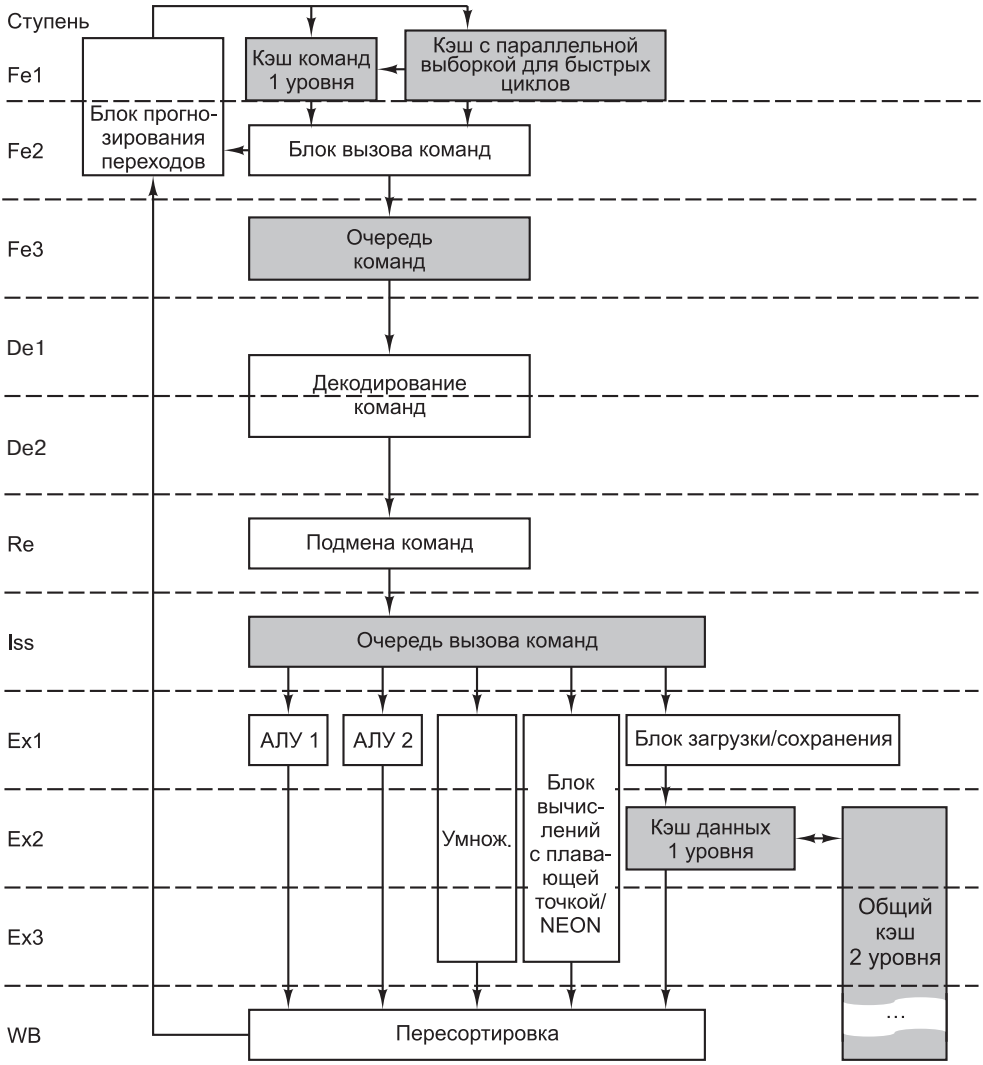


Рис. 4.34. Упрощенная схема конвейера Cortex A9 в OMAP4430

На ступенях *De1* (Decode 1) и *De2* происходит декодирование выбранных команд. Ступени определяют, какие входные данные понадобятся командам (регистры и память) и какие ресурсы потребуются для их исполнения (функциональные блоки). После завершения декодирования команды передаются на ступень *Re* (Rename), на которой происходит подмена регистров для устранения конфликтов WAR и WAW при исполнении с изменением последовательности. На этой ступени находится таблица подмены с информацией о том, какой физический регистр в настоящее время содержит все архитектурные регистры. При помощи этой таблицы можно легко заменить любой входной регистр. Выходным регистрам должен быть предоставлен новый физический

регистр, который берется из пула неиспользованных физических регистров. Назначенный физический регистр будет использоваться командой вплоть до ее завершения.

Далее команды поступают на ступень *Iss* (Instruction Issue), в которой они помещаются в очередь вызова команд. Очередь вызова следит за тем, для каких команд готовы входные данные. В случае готовности ступень получает входные регистровые данные (из физического регистрового файла или обходной шины), после чего команда передается на исполнительные ступени. Как и Core i7, Cortex A9 может выдавать команды в порядке, отличном от порядка их следования в программе. За каждый цикл может выдаваться до четырех команд. Выбор команд ограничивается доступностью функциональных блоков.

Фактическое исполнение команд происходит на ступенях *Ex* (Execute). Большинство арифметических и логических команд, а также команд сдвига используют целочисленные АЛУ и завершаются за один цикл. Загрузка и сохранение занимают два цикла (при попадании в кэш L1), а умножение — три цикла. Ступени *Ex* содержат несколько функциональных блоков:

1. АЛУ 1 для целочисленных операций.
2. АЛУ 2 для целочисленных операций.
3. Блок умножения.
4. АЛУ для операций с плавающей точкой и векторных операций SIMD (возможно, с поддержкой VFP и NEON).
5. Блок загрузки и хранения.

Команды условного перехода также обрабатываются на первой ступени *Ex*, при этом определяется наличие/отсутствие ветвления. Если прогноз окажется ошибочным, сигнал возвращается на ступень *Fe1* и конвейер аннулируется.

После того как исполнение будет завершено, команды входят на ступень WB (WriteBack), где каждая команда непосредственно обновляет физический регистровый файл. Теоретически позднее, когда команда окажется самой старой из выполняемых, она запишет свой регистровый результат в архитектурный регистровый файл. Если произойдет исключение или прерывание, видимыми станут именно эти значения, а не те, которые хранятся в физических регистрах. Акт сохранения регистра в архитектурном файле эквивалентен пересортировке в Core i7. Кроме того, на ступени WB все команды сохранения завершают запись своих результатов в кэш данных L1.

Наше описание Cortex A9 не отличается полнотой, но оно дает некоторое представление о принципах ее работы и отличиях от микроархитектуры Core i7.

Микроархитектура микроконтроллера ATmega168

Схема микроархитектуры из нашего последнего примера — Atmel ATmega168 — изображена на рис. 4.35. Она существенно проще микроархитектур Core i7 и OMAP4430. Дело в том, что микросхема, предназначенная для рынка встроенных систем, должна быть очень маленькой и недорогой. Соответственно главной целью при проектировании ATmega168 была дешевизна, а не скорость. Как известно, «дешевый» и «простой» — очень близкие понятия, в то время как дешевизна и быстродействие в нашем контексте редко сочетаются.

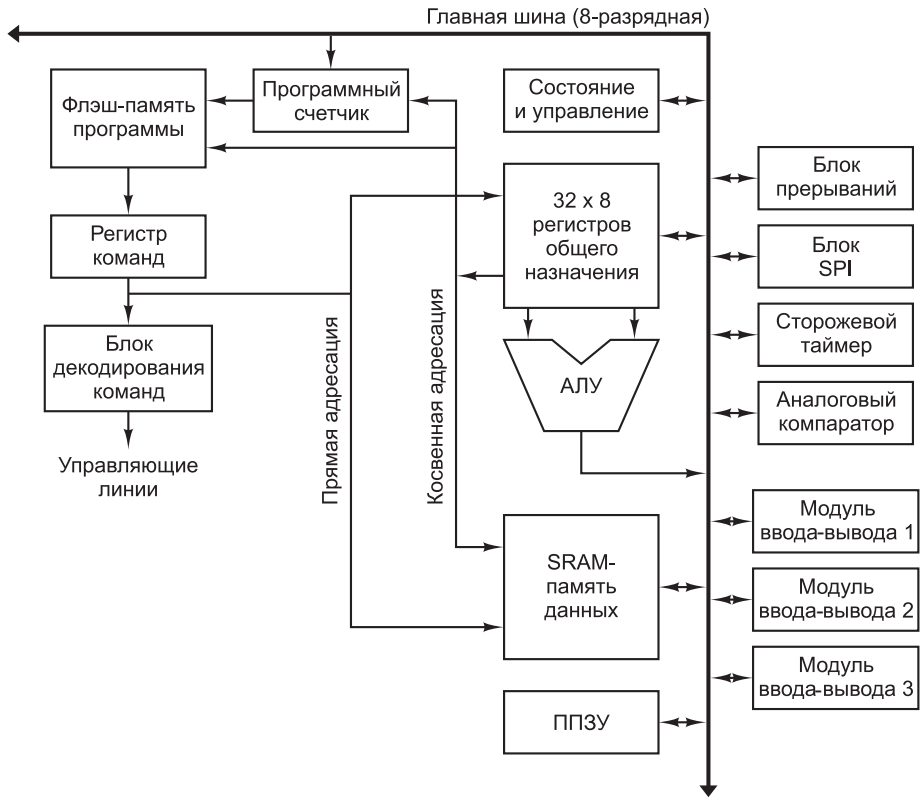


Рис. 4.35. Микроархитектура Atmel ATmega168

Центральное положение в микроархитектуре ATmega168 занимает 8-разрядная основная шина. С ней связаны регистры и биты состояния, АЛУ, память и устройства ввода-вывода. Давайте кратко опишем их сейчас. Регистровый файл состоит из 32 8-разрядных регистров, используемых для хранения временных значений. В **регистре состояния** и **управляющем регистре** содержатся признаки последней операции АЛУ (знак, переполнение, отрицательность, нуль, перенос), а также бит незавершенного прерывания. **Программный счетчик** содержит адрес команды, выполняемой в настоящий момент. Чтобы выполнить операцию АЛУ, необходимо сначала прочитать операнды из регистров и передать их АЛУ. Выходные данные АЛУ записывается в любые регистры с возможностью записи через главную шину.

ATmega168 использует несколько видов памяти для команд и данных. Размер статической памяти данных составляет 1 Кбайт — слишком много для полноценной адресации 8-разрядным адресом на главной шине. По этой причине архитектура AVR позволяет конструировать адреса из последовательной пары 8-разрядных регистров; результат представляет собой 16-разрядный адрес, поддерживающий до 64 Кбайт памяти данных. EEPROM предоставляет до 1 Кбайт энергонезависимой памяти, в которую программы могут записывать переменные, «переживающие» возможное отключение питания.

Аналогичный механизм существует и для адресации программной памяти, но 64 Кбайт — слишком мало для кода, даже в дешевых встроенных системах. Для адресации памяти команд большего объема архитектура AVR определяет три страничных регистра (RAMPX, RAMPY и RAMPZ), размер каждого регистра равен 8 битам. Страничный регистр объединяется с 16-разрядной парой регистров для формирования 24-разрядного программного адреса, что позволяет адресовать до 16 Мбайт адресного пространства команд.

Задумайтесь на секунду: 64 Кбайт кода слишком мало для микроконтроллера, который может обслуживать игрушку или маленькое электронное устройство. В 1964 году фирма IBM выпустила компьютер System 360 Model 30, имевший 64 Кбайт общей памяти (без возможности ее расширения). Компьютер продавался за $250 000, что эквивалентно приблизительно двум миллионам в сегодняшних ценах. ATmega168 стоит около $1 — и меньше при большом объеме партии. Если свериться с ценами на самолеты Boeing, нетрудно заметить, что они никак не сократились в 250 000 раз за последние 50 лет. То же самое можно сказать о машинах, телевизорах и прочей технике — обо всем, кроме компьютеров.

Кроме того, ATmega168 содержит встроенный контроллер прерываний, интерфейс последовательного порта (SPI) и таймеры, необходимые для приложений реального времени. Также имеются три 8-разрядных цифровых порта ввода-вывода, при помощи которых ATmega168 может управлять до 24 внешних кнопок, индикаторов, датчиков и т. д. Именно наличие таймеров и портов ввода-вывода дает возможность использования ATmega168 во встроенных системах без дополнительных микросхем.

Процессор ATmega168 относится к категории синхронных — большинство команд, которые он обрабатывает, завершаются за один цикл. Процессор имеет конвейерную архитектуру: в процессе выборки одной команды происходит выполнение предыдущей команды. Впрочем, конвейер состоит всего из двух ступеней — выборки и исполнения. Чтобы выборка команд производилась за один цикл, цикл должен вмещать чтение регистра из регистрового файла, исполнение команды в АЛУ и последующую запись регистра обратно в регистровый файл. Так как все эти операции происходят за один цикл, логика обхода или обнаружения приостановки не нужна. Команды программы исполняются по порядку, за один цикл и без перекрытия с другими командами.

Об устройстве ATmega168 можно было бы рассказать подробнее, однако имеющегося описания и схемы, изображенной на рис. 4.35, вполне достаточно для того, чтобы получить общее представление. ATmega168 имеет одну основную шину (что позволяет уменьшить размер микросхемы), гетерогенного набора регистров, а также нескольких видов памяти и устройств ввода-вывода, подключенных к основной шине. В течение каждого цикла тракта данных два операнда читаются из регистрового файла, проводятся через АЛУ, а результаты снова сохраняются в регистрах, как и на более современных компьютерах.

Сравнение процессоров i7, OMAP4430 и ATmega168

Приведенные три процессора во многом отличаются друг от друга, однако у них есть удивительная общность, которая может помочь в разработке компьютера. Core i7 поддерживает унаследованный набор CISC-команд, который инженеры

компании Intel рады бы слить в ближайший водоем, но этим бы они безусловно нарушили природоохранное законодательство. OMAP4430 — «чистая» RISC-система с эффективным набором команд. ATmega168 — простой 8-разрядный процессор для встроенных систем. В основе всех представленных примеров — набор регистров и один или несколько АЛУ, исполняющих простые арифметические и булевы операции с операндами из регистров.

Несмотря на различия, Core i7 и OMAP4430 имеют сходные функциональные блоки. У обоих функциональные блоки получают микрооперации, в которых содержится код операции, а также указаны два входных и один выходной регистр. Оба могут выполнять микрооперацию за один цикл. Оба конвейеризированы, поддерживают прогнозирование переходов и разделенную кэш-память для команд и для данных.

Такое внутреннее сходство не случайно, причиной его являются вовсе не постоянные переходы инженеров из одной компании Кремниевой долины в другую. Когда мы рассматривали микроархитектуры Mic-3 и Mic-4, мы видели, что достаточно просто построить конвейеризированный тракт данных с двумя регистрами в качестве источников, в котором значения этих регистров проходят через АЛУ, а результат сохраняется в регистре. На рис. 4.22 представлено графическое изображение такого конвейера. Для современной технологии это наиболее эффективная система.

Главное отличие между Core i7 и OMAP4430 состоит в том, как передаются ISA-команды функциональному блоку. Компьютеру Core i7 приходится разбивать CISC-команды, чтобы преобразовать их в 3-регистровый формат, необходимый для функционального блока. Именно этот процесс показан на рис. 4.32 — разбиение больших команд на маленькие микрооперации. Системе OMAP4430 не нужно ничего делать, поскольку ее исходные команды уже представляют собой удобные и компактные микрооперации. Вот почему большинство новых архитектур ISA относятся к типу RISC, который обеспечивает оптимальное сочетание набора команд и внутреннего механизма их исполнения.

Полезно сравнить нашу последнюю разработку, микроархитектуру Mic-4, с этими двумя реальными машинами. Mic-4 больше напоминает Core i7. Обе системы интерпретируют команды, не являющиеся RISC-командами. Для этого обе системы разбивают команды на микрооперации, в которых указаны код операции, два входных и один выходной регистр. В обоих случаях микрооперации помещаются в очередь для дальнейшего выполнения. В Mic-4 микрооперации запускаются строго по порядку, выполняются строго по порядку и завершаются тоже строго по порядку. В Core i7 микрооперации запускаются по порядку, выполняются в произвольном порядке, а завершаются опять-таки по порядку.

Сравнивать Mic-4 и OMAP4430 некорректно, поскольку команды системы OMAP4430 — это RISC-команды (то есть 3-регистровые микрооперации). Их не нужно ни разбивать, ни объединять. Их можно выполнять в том виде, в котором они есть, каждую за один цикл тракта данных.

По сравнению с Core i7 и OMAP4430, машина ATmega168 очень проста. Ее микроархитектура больше напоминает RISC, чем CISC, поскольку простые команды исполняются в ней за один цикл без разбивки на составные части. Ни конвейеризация, ни кэширование в ней не предусмотрены. Команды запускаются, исполняются и возвращают результаты по порядку. По своей простоте ATmega168 напоминает Mic-1.

Краткое содержание главы

Основным компонентом любого компьютера является тракт данных. Он содержит несколько регистров, одну, две или три шины, а также один или несколько функциональных блоков, например АЛУ и схему сдвига. В основном цикле вызываются несколько операндов из регистров и передаются по шинам к АЛУ и другому функциональному блоку на исполнение. После завершения операции результаты вновь сохраняются в регистрах.

Тракт данных может управляться контроллером последовательности, который вызывает микрокоманды из управляющей памяти. Каждая микрокоманда содержит биты, управляющие трактом данных в течение одного цикла. Эти биты определяют, какие операнды нужно выбрать, какую операцию выполнить и что делать с результатами. Кроме того, каждая микрокоманда указывает на следующую микрокоманду (обычно в ней содержится ее адрес). Некоторые микрокоманды изменяют этот базовый адрес с помощью операции ИЛИ.

IJVM — это машина со стековой организацией и с 1-байтными кодами операций, которые помещают слова в стек, выталкивают слова из стека и выполняют различные операции над словами из стека (например, складывают их). В данной главе описывается микропрограмма для микроархитектуры Mic-1. Если добавить блок выборки команд для загрузки команд из потока байтов, то можно устранить большое количество обращений к счетчику команд, и тогда скорость работы машины значительно возрастет.

Существуют множество подходов к организации уровня микроархитектуры: в том числе 2- или 3-шинная структура, кодированные или декодированные поля микрокоманд, наличие или отсутствие предварительной выборки, конвейер с большим или небольшим количеством ступеней и т. д. Mic-1 — это простая машина с программным управлением, последовательным выполнением команд и полным отсутствием параллелизма. Mic-4, напротив, является микроархитектурой с высокой степенью параллелизма и 7-ступенчатым конвейером.

Производительность компьютера можно повысить несколькими способами, основным из которых является использование кэш-памяти. Кэш-память прямого отображения и ассоциативная кэш-память с множественным доступом позволяют ускорить обращения к памяти. Помимо использования кэш-памяти применяют прогнозирование ветвлений (как статическое, так и динамическое), исполнение с изменением последовательности операций и спекулятивное выполнение команд.

У всех трех процессоров, рассматриваемых нами в качестве примеров — Core i7, OMAP4430 и ATmega168, — микроархитектуры неочевидны для программистов, пишущих на языке ассемблера архитектурного уровня. Core i7 реализует сложную схему преобразования ISA-команд в микрооперации, их кэширования, передачи суперскалярному RISC-ядру для исполнения вне исходной последовательности, подмены регистров и применения всех остальных описанных в этой книге приемов, позволяющих разогнать оборудование до максимально возможной скорости. Что же касается процессора OMAP4430, то, если не считать многоступенчатого конвейера, его микроархитектура достаточно проста: запуск и исполнение команд, а также получение их результатов осуществляются без изменения последовательности команд. Процессор ATmega168 прост до крайности — к одной основной шине подключено несколько регистров и одно АЛУ.

Вопросы и задания

1. Какие четыре шага используются процессорами для исполнения команд?
2. На рис. 4.5 регистр шины B кодируется через 4-разрядное поле, а шина C представлена в виде битового отображения. Почему?
3. На рис. 4.5 есть блок «Старший бит». Нарисуйте его схему.
4. Когда в микрокоманде установлено поле JMPC, регистр MBR соединяется операцией ИЛИ с полем NEXT_ADDRESS, чтобы получить адрес следующей микрокоманды. Существуют ли такие обстоятельства, при которых имеет смысл использовать поле JMPC, если в NEXT_ADDRESS находится значение 0x1FF?
5. Предположим, что в листинге 4.1 после оператора `if` имеется следующий оператор:

   ```
   k = 5;
   ```

   Каким будет новый ассемблерный код при условии, что компилятор является оптимизирующим?
6. Напишите два разных варианта трансляции для IJVM следующего оператора на языке Java:

   ```
   i = k + n + 5;
   ```
7. Напишите на языке Java оператор, ставший источником следующего кода для IJVM:

   ```
   ILOAD j
   ILOAD k
   ISUB
   BIPUSH 6
   ISUB
   DUP
   IADD
   ISTORE i
   ```
8. В этой главе мы упомянули, что при трансляции показанного ниже оператора в двоичную форму метка `L2` должна находиться среди младших 256 слов управляющей памяти:

   ```
   if (Z) goto L1; else goto L2
   ```

   А может ли метка `L1` находиться, скажем, в ячейке с адресом 0x40, а `L2` — в ячейке с адресом 0x140? Объясните, почему.
9. В микрокоманде `if_cmpeq3` микропрограммы для Mic-1 значение регистра MDR копируется в регистр H, а в следующей строке от него отнимается значение регистра TOS. Казалось бы, это удобнее записать в одном операторе:

   ```
   if_cmpeq3    Z=TOS-MDR; rd
   ```

   Почему этого не делают?
10. Сколько времени потребуется машине Mic-1, которая работает на частоте 2,5 ГГц, на выполнение следующего Java-оператора:

    ```
    i = j + k;
    ```

    Ответ дайте в наносекундах.

11. Аналогичный вопрос только уже для машины Mic-2, тоже работающей на частоте 2,5 ГГц. Опираясь на полученный результат, ответьте, сколько времени займет выполнение программы на машине Mic-2, если эта программа выполняется на машине Mic-1 за 100 нс?
12. Напишите для Mic-1 микропрограмму, реализующую JVM-команду `POPTWO`. Эта команда удаляет два верхних слова из стека.
13. На машине JVM существуют специальные 1-байтные коды операций для загрузки в стек локальных переменных от 0 до 3, которые используются вместо обычной команды `ILOAD`. Какие изменения нужно внести в машину IJVM, чтобы наилучшим образом использовать эти команды?
14. Команда `ISHR` (целочисленный арифметический сдвиг вправо) поддерживается в JVM, но не поддерживается в IJVM. Команда извлекает два верхних слова из стека и заменяет их одним словом (результатом). Второе сверху слово стека — это сдвигаемый операнд. Он сдвигается вправо на значение от 0 до 31 включительно, в зависимости от значения пяти самых младших битов верхнего слова в стеке (остальные 27 бит игнорируются). Знаковый бит дублируется вправо на столько же битов, на сколько осуществляется сдвиг. Код операции для команды `ISHR` равен 122 (0x7A).
    1) Какая арифметическая операция эквивалентна сдвигу вправо на два разряда?
    2) Доработайте микрокод, чтобы эта команда поддерживалась в IJVM.
15. Команда `ISHL` (целочисленный сдвиг влево) поддерживается в JVM, но не поддерживается в IJVM. Команда извлекает два верхних слова из стека и заменяет их одним значением (результатом). Второе сверху слово в стеке — сдвигаемый операнд. Он сдвигается влево на значение от 0 до 31 включительно, в зависимости от значения пяти младших битов верхнего слова в стеке (остальные 27 бит верхнего слова игнорируются). Нули сдвигаются влево на столько же битов, на сколько осуществляется сдвиг. Код операции `ISHL` равен 120 (0x78).
    1) Какая арифметическая операция эквивалентна сдвигу влево на два разряда?
    2) Доработайте микрокод, чтобы эта команда поддерживалась в IJVM.
16. JVM-команде `INVOKEVIRTUAL` нужно знать, сколько у нее параметров. Зачем?
17. Реализуйте JVM-команду `DLOAD` для Mic-2. Эта команда содержит 1-байтный индекс и помещает локальную переменную, находящуюся в этом месте, в стек. Затем она помещает следующее старшее слово в стек.
18. Нарисуйте конечный автомат для учета очков при игре в теннис. Правила игры в теннис следующие. Чтобы выиграть, вам нужно получить как минимум 4 очка и у вас должно быть как минимум на 2 очка больше, чем у вашего соперника. Начните с состояния (0, 0), то есть с того, что ни у кого из вас еще нет очков. Затем добавьте состояние (1, 0), означающее, что игрок *A* получил очко. Ребро от состояния (0, 0) к состоянию (1, 0) обозначьте буквой *A*. Затем добавьте состояние (0, 1), чтобы показать, что игрок *B* получил очко, а ребро

к состоянию (0, 1) обозначьте буквой *B*. Продолжайте добавлять состояния и ребра до тех пор, пока не нарисуете все возможные состояния.

19. Вернитесь к предыдущему вопросу. Существуют ли такие состояния, которые можно безболезненно удалить, никак не повлияв на результат любой игры? Если да, то какие из них эквивалентны?
20. Нарисуйте конечный автомат для прогнозирования переходов — более надежный, чем тот, который изображен на рис. 4.29. Он должен изменять прогноз только после трех последовательных неудачных прогнозов.
21. Сдвиговый регистр, изображенный на рис. 4.18, имеет максимальную емкость 6 байт. Можно ли сконструировать блок выборки команд с 5-байтным сдвиговым регистром? А с 4-байтным?
22. Предыдущий вопрос связан с удешевлением блока выборки команд. Теперь рассмотрим вопрос его удорожания. Может ли когда-нибудь понадобиться сдвиговый регистр большей емкости, скажем, на 12 байт? Если да, то почему? Если нет, то почему?
23. В микропрограмме для микроархитектуры Mic-2 микрокоманда `if_icmpeq6` выполняет переход к `T`, если бит Z установлен в 1. Однако микрокоманда `T` та же, что и для `goto1`. А возможно ли перейти к `goto1` сразу и станет ли машина работать быстрее после этого?
24. В микроархитектуре Mic-4 блок декодирования отображает IJVM-код операции в индекс ПЗУ, где хранятся соответствующие микрооперации. Кажется, что было бы проще опустить ступень декодирования и сразу передать IJVM-код операции в очередь. Тогда можно использовать IJVM-код операции в качестве индекса в ПЗУ микроопераций, точно так же, как в микроархитектуре Mic-1. Что не так в этом плане?
25. Почему компьютеры оснащаются многоуровневыми кэшами? Разве не лучше создать один большой кэш?
26. Компьютер содержит двухуровневую кэш-память. Предположим, что 60 % обращений к памяти — это кэш-попадания в кэш-память первого уровня, 35 % — в кэш-память второго уровня, а 5 % — это кэш-промахи. Время доступа составляет 5 нс, 15 нс и 60 нс соответственно, причем время доступа в кэш-память второго уровня и в основную память отсчитывается с того момента, как становится известно, что соответствующая память нужна (например, доступ к кэш-памяти второго уровня не может начаться, пока не произойдет кэш-промах при поиске в кэш-памяти первого уровня). Каково среднее время доступа?
27. В конце подраздела «Кэш-память» в разделе «Повышение производительности» было отмечено, что заполнение по записи выгодно только в том случае, если имеют место повторные записи в одну и ту же строку кэша. А если после записи следуют многочисленные считывания из одной и той же строки? Будет ли полезным заполнение по записи преимуществом в этом случае?
28. В черновом варианте этой книги на рис. 4.27 вместо 4-входовой ассоциативной кэш-памяти была изображена 3-входовая ассоциативная кэш-память. Один из рецензентов заявил, что читателей это может смутить, поскольку 3 — это не степень двойки, а компьютеры все делают в двоичной системе.

Поскольку потребитель всегда прав, рисунок изменили на 4-входовую ассоциативную кэш-память. Был ли рецензент прав? Аргументируйте.

29. Проектировщики компьютеров прикладывают значительные усилия к увеличению глубины конвейеров. Почему?
30. Компьютер с пятиступенчатым конвейером после обработки условного перехода простаивает следующие три цикла. Насколько этот простой снизит производительность, если 20 % команд являются командами условного перехода? Другие причины простоя не учитывайте.
31. Предположим, что компьютер осуществляет предварительную выборку до 20 команд. В среднем 4 из них являются командами условного перехода, причем вероятность правильного прогноза каждого составляет 90 %. Какова вероятность, что предварительно вызванная команда находится на одном из правильных путей?
32. Предположим, что нам пришлось изменить структуру машины, представленную с помощью табл. 4.11, чтобы использовать 16 регистров вместо 8. Тогда мы изменим команду 6, чтобы регистр R8 стал целевым. Что в этом случае будет происходить в циклах, начиная с цикла 6?
33. Обычно взаимозависимости затрудняют работу конвейеризированных процессоров. Можно ли что-нибудь сделать с WAW-взаимозависимостью, чтобы улучшить положение вещей? Какие существуют средства оптимизации?
34. Перепишите интерпретатор Mic-1 таким образом, чтобы регистр LV указывал на первую локальную переменную, а не на связующий указатель.
35. Напишите моделирующую программу для 1-входовой кэш-памяти прямого отображения. Сделайте число элементов и длину строки параметрами программы. Поэкспериментируйте с этой программой и представьте полученные результаты.

Глава 5
Уровень архитектуры набора команд

В этой главе подробно обсуждается уровень архитектуры набора команд (ISA). Как показано на рис. 1.2, он расположен между уровнями микроархитектуры и операционной системы. Исторически этот уровень развился прежде всех остальных уровней и изначально был единственным. В наши дни этот уровень очень часто называют «архитектурой» машины, а иногда (что неправильно) «языком ассемблера».

Уровень архитектуры набора команд имеет особое значение: он является связующим звеном между программным и аппаратным обеспечением. Конечно, можно было бы сделать так, чтобы аппаратное обеспечение непосредственно выполняло программы, написанные на C, C++, Java или других языках высокого уровня, но это не очень хорошая идея. Преимущество компиляции перед интерпретацией было бы тогда потеряно. Кроме того, из чисто практических соображений компьютеры должны уметь выполнять программы, написанные на разных языках, а не только на одном.

Практически все проектировщики систем считают, что программы, написанные на различных языках высокого уровня, должны транслироваться в некую общую для всех промежуточную форму — уровень архитектуры набора команд; аппаратное обеспечение ориентируется на непосредственное выполнение программ этого уровня. Уровень архитектуры набора команд связывает компиляторы и аппаратное обеспечение. Это язык, который понятен и компиляторам, и устройствам. На рис. 5.1 показана взаимосвязь компиляторов, уровня архитектуры набора команд и аппаратного обеспечения.

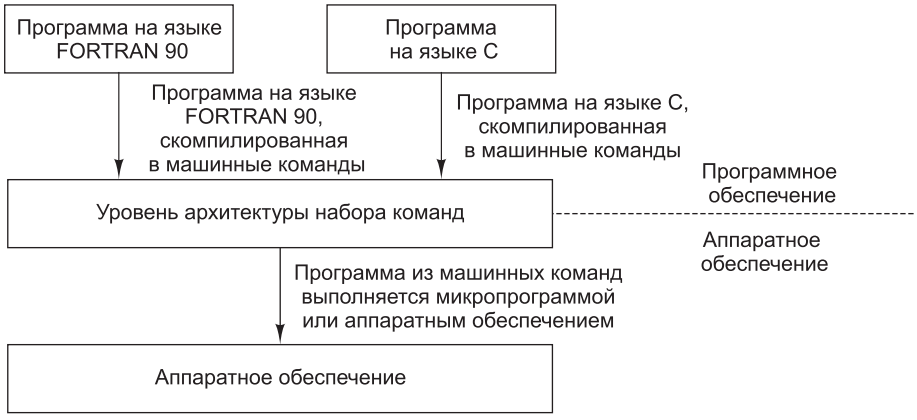


Рис. 5.1. Уровень архитектуры набора команд — это промежуточное звено между компиляторами и аппаратным обеспечением

В идеале при создании новой машины разработчики архитектуры команд должны консультироваться с разработчиками и компиляторов, и аппаратуры, чтобы выяснить, какими особенностями должен обладать подведомственный им уровень. Если разработчики компилятора потребуют какого-то свойства, которое инженеры не смогут реализовать с приемлемым уровнем затрат (скажем, безусловный переход с начислением премии программисту), то такая идея не пройдет. Точно так же, если разработчики аппаратного обеспечения захотят ввести в компьютер какой-либо новый замысловатый элемент (например, память со сверхбыстрыми обращениями по адресам, которые являются простыми числами), но программисты не смогут написать поддерживающую его программу, такой проект никогда не воплотится. Реализуемый в конечном итоге уровень архитектуры набора команд, оптимизированный для нужных языков программирования, всегда является плодом долгих обсуждений и моделирования.

Но все это в теории. А теперь перейдем к суровой реальности. Когда появляется новая машина, первый вопрос, который задают все потенциальные покупатели: «Совместима ли машина с предыдущими версиями?» Второй вопрос: «Можно ли запустить на ней прежнюю операционную систему?» И третий вопрос: «Будут ли работать на этой машине прежние приложения и не потребуется ли заменять их новыми версиями?» Если ответ на любой из этих вопросов оказывается отрицательным, разработчикам придется привести очень убедительные доводы. Покупатели вряд ли захотят выбросить свои любимые программы, чтобы начать все заново.

Этот факт заставляет производителей компьютеров поддерживать один и тот же уровень команд в разных моделях или, по крайней мере, делать его **обратно совместимым**. Под обратной совместимостью мы понимаем способность новой машины выполнять старые программы без изменений. В то же время новая машина может поддерживать новые команды и иметь другие особенности, используемые новым программным обеспечением. Разработчики должны делать уровень команд совместимым с предыдущими моделями, но они вправе как угодно менять аппаратное обеспечение, поскольку едва ли кого-нибудь из покупателей волнует, что собой в реальности представляют «внутренности» компьютера и что именно делает то или иное устройство. Разработчики могут переходить от микропрограмм к непосредственному использованию устройств, добавлять конвейеры, реализовывать суперскалярные схемы и т. п., но при условии, что они сохранят обратную совместимость с уровнем команд предыдущих моделей. Основная цель — убедиться, что старые программы работают на новой машине. То есть на первый план выходит задача не просто создания хороших машин, а создания хороших машин при условии их обратной совместимости.

Все сказанное вовсе не принижает значимость уровня архитектуры набора команд. Качественный уровень архитектуры набора команд чрезвычайно важен особенно в отношении вычислительных возможностей и стоимости. Производительность эквивалентных машин с разными уровнями архитектуры набора команд может отличаться на 25 %. Мы просто хотим сказать, что рынок в определенной степени затрудняет переход от старой архитектуры команд к новой. Тем не менее иногда появляются новые уровни команд универсального назначения, а на специализированных рынках (например, на рынке встроенных

систем или на рынке мультимедийных процессоров) они возникают гораздо чаще. Следовательно, важно понимать принципы работы этого уровня.

Какую архитектуру команд можно считать хорошей? Существует два основных фактора. Во-первых, хорошая архитектура должна определять набор команд, которые можно эффективно реализовать не только в современной, но и в будущей технике. При плохо разработанной архитектуре команд процессор может строиться из большего числа вентилей, для выполнения программ может требоваться больший объем памяти и т. д. Он также может медленнее работать, потому что архитектура набора команд скроет возможности параллельного выполнения операций, и для достижения эквивалентной производительности потребуются более сложные схемы. Проект, в котором должным образом учтены особенности конкретной техники, может стать основой для производства целого поколения компьютеров, прийти на смену которым сможет система с еще более совершенной архитектурой команд.

Во-вторых, хорошая архитектура команд должна обеспечивать предельную ясность в отношении того, какой именно должна быть откомпилированная программа. Четкость и полнота вариантов — те черты, которые не всегда свойственны архитектуре команд. Эти черты особенно важны для компилятора, который не всегда может сделать оптимальный выбор из нескольких альтернатив, особенно если некоторые очевидные на первый взгляд альтернативы архитектурой команд не поддерживаются. Если говорить кратко, поскольку уровень архитектуры набора команд является промежуточным звеном между аппаратным и программным обеспечением, он должен устраивать как разработчиков аппаратного обеспечения (с точки зрения эффективной реализации), так и программистов (с точки зрения простоты генерирования качественного кода).

Общий обзор уровня архитектуры набора команд

Давайте начнем изучение уровня архитектуры набора команд с вопроса о том, что он собой представляет. Этот вопрос на первый взгляд может показаться простым, но на самом деле здесь очень много тонкостей. В следующем подразделе мы обсудим некоторые из них, а затем рассмотрим модели памяти, регистров и команд.

Свойства уровня архитектуры набора команд

В принципе, уровень архитектуры набора команд — это тот уровень, на котором компьютер представляется программисту, пишущему программы на машинном языке. Поскольку сейчас ни один нормальный программист таких программ не пишет, мы слегка переделали это определение: программа уровня архитектуры набора команд — это то, что получается в результате работы компилятора (в данный момент мы не будем касаться системных вызовов и символического языка ассемблера). Чтобы получить программу уровня архитектуры набора команд, создатель компилятора должен знать, какая модель памяти используется

в машине, какие регистры, типы данных и команды имеются в наличии и т. д. Вся эта информация в совокупности и определяет уровень архитектуры набора команд.

В соответствии с этим определением вопросы о том, доступна ли микроархитектура программно, конвейеризирован ли компьютер, является ли он суперскалярным и т. д., не относятся к уровню архитектуры набора команд, поскольку разработчик компилятора всего этого не видит. Впрочем, это замечание истинно лишь отчасти; некоторые из этих характеристик влияют на производительность — показатель, вполне доступный для разработчика компилятора. Рассмотрим, например, суперскалярную машину, которая в одном цикле может обрабатывать сдвоенные команды, причем такие, что одна команда целочисленная, а вторая с плавающей точкой. Если в коде, полученном в результате работы компилятора, целочисленные команды и команды с плавающей точкой будут чередоваться, то производительность заметно повысится. Таким образом, детали суперскалярной операции *доступны* на уровне архитектуры набора команд, то есть границы между разными уровнями размыты.

Для одних архитектур уровень команд определяется формальным документом, который обычно выпускается промышленным консорциумом, для других — нет. Например, ARM v7 (версия 7 ARM ISA) имеют официальное определение, опубликованное ARM Ltd. Цель такого официального документа — дать возможность различным производителям выпускать машины данного типа, чтобы эти машины могли выполнять одни и те же программы и получать при этом одни и те же результаты.

В случае с ARM подобные документы нужны для того, чтобы различные предприятия могли выпускать идентичные микросхемы ARM, отличающиеся друг от друга только по производительности и цене. Чтобы эта идея работала, поставщики микросхем должны знать, что делает микросхема ARM (на уровне архитектуры набора команд). Следовательно, в документе говорится о том, какова модель памяти, какие имеются регистры, какие действия выполняют команды и т. д., а не о том, что представляет собой микроархитектура.

В таких документах содержатся **нормативные** разделы, в которых излагаются требования, и **информативные** разделы, призванные помочь читателю, но не являющиеся частью формального определения. В нормативных разделах постоянно встречаются такие слова, как *должно быть*, *нельзя*, *следует*, означающие соответственно требование, запрет и рекомендацию. Например, следующее предложение означает, что если программа выполняет код операции, который не определен, он должен вызывать системное прерывание, а не просто игнорироваться:

Выполнение зарезервированного кода операции должно вызывать системное прерывание.

Может быть и альтернативный подход:

Результат выполнения зарезервированного кода операции определяется реализацией.

Это значит, что разработчик компилятора не может рассчитывать на какое-то конкретное поведение; следовательно, у разработчиков появляется свобода выбора. К описанию архитектуры часто прилагаются тестовые пакеты для проверки, действительно ли данная реализация соответствует техническим требованиям.

Совершенно ясно, почему ARM v7 поставляется с документом, в котором определяется уровень архитектуры набора команд — это нужно для того, чтобы все микросхемы ARM могли выполнять одни и те же программы. Многие годы формального определяющего документа для архитектуры набора команд IA-32 (или **x86**) не существовало — компания Intel не хотела, чтобы другие производители могли выпускать Intel-совместимые микросхемы. Она даже обращалась в суд, чтобы запретить производство своих микросхем другими производителями, но проиграла процесс. Однако в конце 1990-х компания Intel все же опубликовала полную спецификацию набора команд IA-32. То ли она осознала ошибочность своего подхода и захотела помочь коллегам-проектировщикам и программистам, то ли дело было в том, что США, Япония и Европа решили проверить Intel на нарушение монопольного законодательства. Хорошо написанный справочник по набору команд Intel находится и продолжает обновляться на веб-сайте Intel для разработчиков (*http://developer.intel.com*). Объем версии с командами Intel Core i7 составляет 4161 страницу; это лишний раз напоминает нам о том, что Core i7 относится к компьютерам со *сложным* набором команд.

Другое важное качество уровня архитектуры набора команд состоит в том, что в большинстве машин поддерживаются, по крайней мере, два режима. В **привилегированном режиме** запускается операционная система. Этот режим позволяет выполнять все команды. **Пользовательский режим** предназначен для запуска прикладных программ. Он не позволяет выполнять некоторые потенциально опасные команды (например, те, которые непосредственно манипулируют кэш-памятью). В этой главе мы в первую очередь сосредоточимся на командах и свойствах пользовательского режима.

Модели памяти

Во всех компьютерах память разделена на ячейки, которые имеют последовательные адреса. В настоящее время наиболее распространенный размер ячейки — 8 бит, но раньше использовались ячейки от 1 до 60 бит (см. табл. 2.1). Ячейка из 8 бит называется **байтом** (или **октетом**). Причина применения именно 8-разрядных ячеек памяти состоит в том, что ASCII-символ занимает 7 бит, поэтому он (вместе с редко используемым битом четности) умещается в одном байте. В других кодировках — таких, как Unicode и UTF-8 — для представления символов используются группы битов, кратные 8.

Байты обычно группируются в 4-байтные (32-разрядные) или 8-байтные (64-разрядные) слова, а в наборе присутствуют команды манипулирования целыми словами. Многие архитектуры требуют, чтобы слова выравнивались по своим естественным границам. Так, 4-байтное слово может начинаться с адреса 0, 4, 8 и т. д., но не с адреса 1 или 2. Точно так же слово из 8 байт может начинаться с адреса 0, 8 или 16, но не с адреса 4 или 6. Механизм размещения 8-байтных слов в памяти иллюстрирует рис. 5.2.

Выравнивание адресов требуется довольно часто, поскольку при этом память работает наиболее эффективно. Например, процессор Core i7, который вызывает из памяти по 8 байт через интерфейс DDR3, поддерживает только выровненные 64-разрядные обращения. Следовательно, Core i7 вообще не сможет обратиться к невыровненной памяти, поскольку интерфейс памяти требует, чтобы адреса были кратны 8.

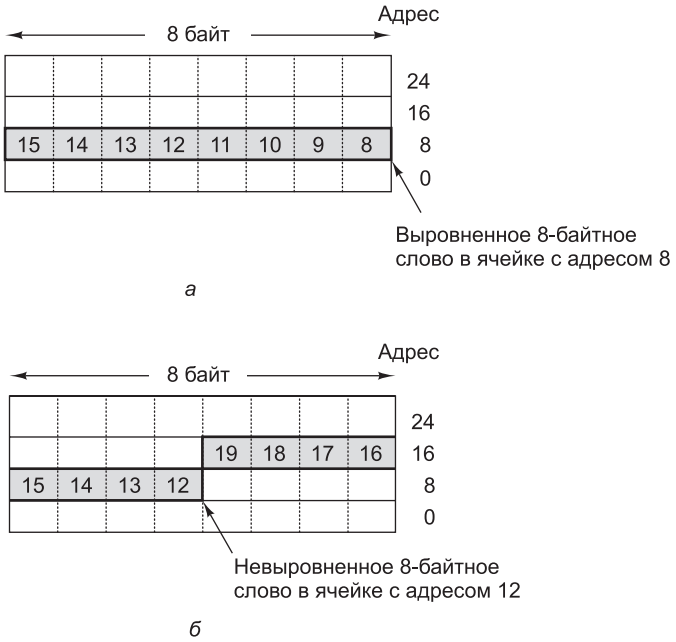


Рис. 5.2. Расположение слова из 8 байт в памяти: выровненное слово (*а*); невыровненное слово (*б*). Некоторые машины требуют, чтобы слова в памяти были выровнены

Однако требование относительно выравнивания адресов иногда вызывает некоторые проблемы. В процессоре Core i7 программы могут обращаться к словам, начиная с любого адреса, — это качество восходит к модели 8088 с шиной данных шириной 1 байт, в которой не требовалось, чтобы ячейки располагались в 8-байтных границах. Если программа в процессоре Core i7 считывает 4-байтное слово с адреса 7, аппаратное обеспечение должно сделать одно обращение к памяти, чтобы вызвать байты с 0 по 7, а второе — чтобы вызвать байты с 8 по 15. Затем центральный процессор извлекает требуемые 4 байта из 16, считанных из памяти, и компонует их в нужном порядке, чтобы сформировать 4-байтное слово. Понятно, что частое выполнение таких операций плохо сочетается с высокой производительностью.

Возможность считывать слова с произвольными адресами требует усложнения микросхемы, которая после этого становится больше и дороже. Разработчики были бы рады избавиться от такой микросхемы и просто потребовать, чтобы все программы обращались к памяти пословно, а не побайтно. Однако на традиционный вопрос разработчиков: «Кому нужны древние программы, написанные еще для машин 8088 и совершенно неправильно работающие с памятью?» — следует не менее традиционный ответ продавцов: «Нашим клиентам».

Большинство машин имеют единое линейное адресное пространство, которое простирается от адреса 0 до какого-то максимума, обычно 2^{32} или 2^{64} байт. В некоторых машинах содержатся раздельные адресные пространства для команд и данных, так что при вызове команды с адресом 8 и вызове данных с адресом 8 происходит обращение к разным адресным пространствам. Такая система гораздо

сложнее, чем единое адресное пространство, но зато она имеет два преимущества. Во-первых, все с теми же 32-разрядными адресами появляется возможность иметь 2^{32} байт для программ и дополнительные 2^{32} байт для данных. Во-вторых, поскольку запись всегда автоматически происходит только в пространство данных, случайная перезапись программы становится невозможной, и следовательно, устраняется один из источников программных ошибок. Наконец, разделение команд и данных усложняет атаки со стороны вредоносных программ, которые не могут изменить программный код — они даже не могут адресовать его.

Отметим, что раздельные адресные пространства для команд и для данных — это не то же самое, что разделенная кэш-память первого уровня. В первом случае все адресное пространство целиком дублируется, и считывание из любого адреса вызывает разные результаты в зависимости от того, что именно считывается: слово данных или команда. При разделенной кэш-памяти существует только одно адресное пространство, просто в разных блоках кэш-памяти хранятся разные части этого пространства.

Еще один аспект модели памяти — семантика памяти. Естественно ожидать, что команда `LOAD`, если она выполняется после команды `STORE`, обратится к тому же адресу и возвратит только что сохраненное значение. Однако, как мы видели в главе 4, во многих машинах микрокоманды переупорядочиваются. Таким образом, существует реальная опасность, что память будет работать не так, как ожидается. Ситуация усложняется при наличии мультипроцессора, когда каждый процессор посылает в общую память поток запросов на чтение и запись, и эти запросы тоже могут быть переупорядочены.

Проектировщики систем используют несколько вариантов решения этой проблемы. С одной стороны, все запросы к памяти могут быть упорядочены таким образом, чтобы каждый из них завершался до того, как начнется следующий. Такая стратегия отрицательно сказывается на производительности, но зато дает простейшую семантику памяти (все операции выполняются строго в том порядке, в котором они расположены в программе).

С другой стороны, можно вообще не давать никаких гарантий относительно упорядоченности запросов к памяти, а чтобы добиться такой упорядоченности, программа выполняет команду `SYNC`, которая блокирует запуск всех новых операций с памятью до тех пор, пока не завершатся предыдущие. Эта идея весьма затрудняет работу создателей компиляторов, поскольку им приходится тщательно разбираться в том, как работает соответствующая микроархитектура, но зато разработчикам аппаратного обеспечения предоставлена полная свобода в плане оптимизации использования памяти.

Возможны также промежуточные модели памяти, в которых аппаратное обеспечение автоматически блокирует запуск определенных операций с памятью (например, тех, которые связаны с RAW- и WAR-взаимозависимостями), при этом запуск всех других операций не блокируется. Хотя реализация этих возможностей на уровне архитектуры набора команд довольно утомительна (по крайней мере, для создателей компиляторов и программистов на языке ассемблера), сейчас заметна тенденция к преобладанию подобного подхода. Данная тенденция вызвана к жизни такими разработками, как механизмы переупорядочения микрокоманд, конвейеры, многоуровневая кэш-память и т. д. Другие, менее известные, примеры такого рода мы рассмотрим в этой главе чуть позже.

Регистры

Во всех компьютерах имеются несколько регистров, доступных на уровне архитектуры набора команд. Они позволяют управлять выполнением программы, хранить временные результаты, а также используются для других целей. Обычно регистры, доступные на уровне микроархитектуры (например, TOS и MAR на рис. 4.1), на уровне архитектуры набора команд недоступны, однако некоторые регистры, например счетчик команд и указатель стека, доступны на обоих уровнях. В то же время регистры, доступные на уровне архитектуры набора команд, всегда доступны на уровне микроархитектуры, поскольку именно там они реализованы.

Регистры уровня архитектуры набора команд можно разделить на две категории: специальные регистры и регистры общего назначения. К специальным регистрам относятся счетчик команд и указатель стека, а также другие регистры, имеющие особые функции. Регистры общего назначения содержат ключевые локальные переменные и промежуточные результаты вычислений. Их основная функция состоит в том, чтобы обеспечить быстрый доступ к часто используемым данным (обычно без обращений к памяти). RISC-машины с высокоскоростными процессорами и (относительно) медленной памятью обычно содержат как минимум 32 регистра общего назначения, причем в новых процессорах количество регистров общего назначения постоянно растет.

В некоторых машинах регистры общего назначения полностью симметричны и взаимозаменяемы. Если все регистры эквивалентны, для хранения промежуточного результата компилятор может использовать как регистр R1, так и регистр R25. Выбор регистра не имеет никакого значения.

В других машинах некоторые регистры общего назначения могут быть специализированными. Например, в процессоре Core i7 имеется регистр EDX, который может использоваться в качестве регистра общего назначения, но который, кроме того, используется для решения сугубо специфических задач (получает половину произведения при умножении и половину делимого при делении).

Даже если регистры общего назначения полностью взаимозаменяемы, операционная система или компиляторы часто соблюдают формальные соглашения по их использованию. Например, некоторые регистры могут применяться для хранения параметров вызываемых процедур, другие исполняют роль временных. Если компилятор помещает важную локальную переменную в регистр R1, а затем вызывает библиотечную процедуру, которая воспринимает R1 как регистр, временно выделенный в ее распоряжение, то после возвращения значения этой процедурой в регистре R1 может остаться «мусор». То есть если существуют какие-либо системные соглашения по поводу того, как нужно использовать регистры, разработчики компиляторов и программисты, пишущие на ассемблере, должны им следовать.

Помимо регистров, доступных на уровне архитектуры набора команд, всегда существуют довольно много специальных регистров, доступных только в привилегированном режиме. Эти регистры управляют различными блоками кэш-памяти, основной памятью, устройствами ввода-вывода, другими устройствами машины. Данные регистры используются только операционной системой, поэтому компиляторам и пользователям не обязательно знать об их существовании.

Один «гибридный» регистр доступен и в привилегированном, и в пользовательском режимах. Это **регистр PSW** (Program State Word — **слово состояния программы**), который еще называют **флаговым**. Флаговый регистр содержит различные биты, необходимые центральному процессору. Самые важные биты — это **коды условий**. Они устанавливаются в каждом цикле АЛУ и отражают состояние результата предыдущей операции:

- N — результат отрицателен (Negative);
- Z — результат равен 0 (Zero);
- V — результат вызвал переполнение (oVerflow);
- C — перенос самого левого бита (Carry out);
- A — перенос бита 3 (Auxiliary carry — служебный перенос);
- P — результат четный (Parity).

Коды условий очень важны, поскольку используются при сравнениях и условных переходах. Например, команда `CMP` обычно вычитает один операнд из другого и на основе полученной разности устанавливает коды условий. Если операнды равны, то разность будет равна 0, а во флаговом регистре установится бит Z. Последующая команда `BEQ` (Branch Equal — переход в случае равенства) проверяет бит Z и совершает переход, если он установлен.

Флаговый регистр может хранить не только коды условий. Его содержимое в разных машинах может быть разным. Дополнительные поля указывают режим машины (например, пользовательский или привилегированный), бит трассировки (который используется для отладки), уровень приоритета процессора, статус разрешения прерываний. Флаговый регистр обычно читается в пользовательском режиме, но некоторые поля могут записываться только в привилегированном режиме (например, бит, который указывает режим).

Команды

Главная особенность уровня, который мы сейчас рассматриваем, — это набор машинных команд. Они управляют действиями машины. В этом наборе всегда в той или иной форме присутствуют команды `LOAD` и `STORE`, предназначенные для перемещения данных между памятью и регистрами, и команда `MOVE`, которая служит для копирования данных из одного регистра в другой. Также всегда присутствуют арифметические и логические команды, команды сравнения элементов данных и команды переходов в зависимости от результатов. Некоторые типичные команды мы уже рассматривали в главе 4 (см. табл. 4.2.), а в этой главе мы познакомимся со многими другими.

Общий обзор уровня архитектуры набора команд Core i7

В этой главе мы обсудим три совершенно разные архитектуры команд: IA-32 компании Intel (она реализована в Core i7), ARM v7 (она реализована в однокристальной системе OMAP4430) и 8-разрядную архитектуру AVR, используемую микроконтроллером ATmega168. Мы не преследуем цель дать исчерпывающее описание каждой из этих архитектур. Мы просто хотим продемонстрировать важные аспекты архитектуры команд и показать, как эти аспекты меняются от одной архитектуры к другой. Начнем с машины Core i7.

Процессор Core i7 развивался на протяжении многих лет. Как отмечалось в главе 1, его история восходит к самым первым микропроцессорам. Основная архитектура команд обеспечивает выполнение программ, написанных для процессоров 8086 и 8088 (которые имеют одну и ту же архитектуру команд), и отчасти даже для 8080 — 8-разрядного процессора, который был популярен в 70-е годы. На 8080, в свою очередь, в значительной степени повлияли требования совместимости с процессором 8008, построенном на базе процессора 4004 (4-разрядной микросхеме, применявшейся еще в каменном веке).

С точки зрения программного обеспечения компьютеры 8086 и 8088 были 16-разрядными (хотя компьютер 8088 содержал 8-разрядную шину данных). Их последователь, 80286, также был 16-разрядным. Его главным преимуществом был больший объем адресного пространства, хотя очень немногие программы его использовали, поскольку оно состояло из 16 384 64-килобайтных сегментов, а не представляло собой линейную 2^{30}-байтную память.

Процессор 80386 был первой 32-разрядной машиной, выпущенной компанией Intel. Все последующие процессоры (80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Pentium M, Centrino, Core 2 duo, Core i7 и т. д.) имеют точно такую же 32-разрядную архитектуру, которая называется **IA-32**, поэтому мы сосредоточим наше внимание именно на этой архитектуре. Единственным существенным изменением архитектуры со времен процессора 80386 было введение в более поздние версии x86 команд MMX, SSE и SSE2. Эти узкоспециализированные команды повышают производительность мультимедийных приложений. Другим важным дополнением стало 64-разрядное расширение x86 (часто называемое x86-64), увеличивающее размеры данных при целочисленных вычислениях и виртуальных адресов до 64 бит. Хотя большинство расширений сначала вводилось компанией Intel, а позднее реализовывалось конкурентами, это расширение было изначально введено компанией AMD.

Core i7 имеет 3 операционных режима, в двух из которых он работает как 8088. В **реальном режиме** все функции, которыми был наделен процессор со времен 8088, отключаются, и Core i7 работает как простой процессор 8088. При программной ошибке происходит полный отказ системы. Если бы компания Intel занималась разработкой человеческих существ, то внутрь каждого такого существа непременно помещался бы специальный бит, возвращающий человека в режим функционирования своих предков (примитивный мозг, отсутствие речи, обитание на деревьях, сугубо банановая диета и т. д.).

На следующей ступени находится **режим виртуального процессора 8086**, который делает возможным исполнение старых программ, написанных для 8088, но с защитой. Чтобы запустить старую программу 8088, операционная система создает специальную изолированную среду, которая работает как процессор 8088, если не считать того, что при программном сбое операционной системе передается соответствующая информация, и полного краха системы не происходит. Когда пользователь Windows открывает окно MS-DOS, запускаемая в этом окне программа выполняется в режиме виртуального процессора 8086 — это позволяет защитить Windows от возможных вольностей DOS-программ.

Последний режим — это **защищенный режим**, в котором Core i7 работает как Core i7, а не как очень дорогой 8088. В этом режиме доступны 4 уровня привилегий, задаваемые битами во флаговом регистре (PSW). Уровень 0 соот-

ветствует привилегированному режиму на других компьютерах и обеспечивает полный доступ к машине. Этот уровень используется операционной системой. Уровень 3 предназначен для пользовательских программ. На этом уровне блокируется доступ к определенным командам и регистрам управления, чтобы сбой какой-нибудь пользовательской программы не привел к краху всей системы. Уровни 1 и 2 применяются редко.

Core i7 имеет огромное адресное пространство. Память разделена на 16 384 сегмента, каждый из которых занимает адреса от 0 до $2^{32} - 1$. Однако большинство операционных систем (включая UNIX и все версии Windows) поддерживают только один сегмент, поэтому для прикладных программ обычно доступно линейное адресное пространство размером 2^{32} байт, причем иногда часть этого пространства занимает сама операционная система. Каждый байт в адресном пространстве имеет свой адрес. Слова состоят из 32 бит. Байты нумеруются справа налево (то есть самый первый адрес соответствует самому младшему байту).

Регистры процессора Core i7 показаны на рис. 5.3. Первые четыре регистра EAX, EBX, ECX и EDX — 32-разрядные. Это регистры общего назначения, хотя

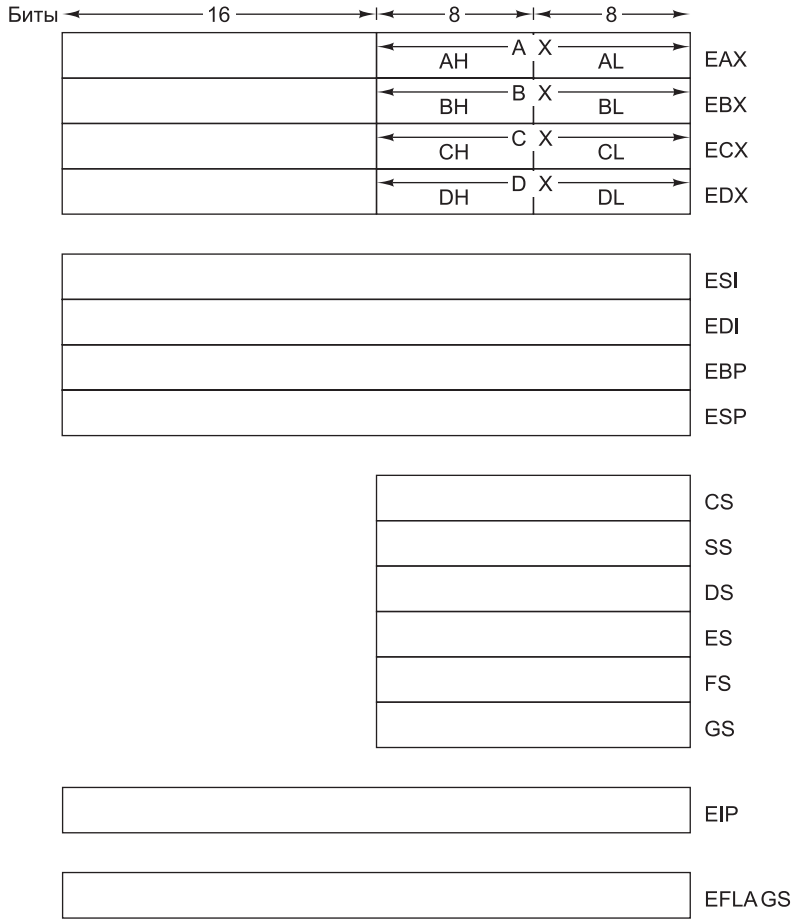


Рис. 5.3. Основные регистры процессора Core i7

у каждого из них есть определенные особенности. EAX — основной арифметический регистр; EBX предназначен для хранения указателей (адресов памяти); ECX связан с организацией циклов; EDX нужен для умножения и деления — этот регистр вместе с EAX содержит 64-разрядные произведения и делимые. Младшие 16 и 8 бит в каждом из рассматриваемых регистров — это самостоятельные 16- и 8-разрядные регистры соответственно, позволяющие легко манипулировать 16- и 8-разрядными значениями. В компьютерах 8088 и 80286 имеются только 8- и 16-разрядные регистры, 32-разрядные регистры появились в системе 80386 вместе с приставкой E (Extended — расширенный).

Следующие три регистра также являются регистрами общего назначения, но с большей степенью специализации. Регистры ESI и EDI предназначены для хранения указателей и в основном ориентированы на аппаратную поддержку строковых команд: ESI указывает на исходную строку, EDI — на целевую. Регистр EBP тоже предназначен для хранения указателей и обычно используется для указания на базу текущего кадра локальных переменных, как и регистр LV в машине IJVM. Такой регистр обычно называют **указателем кадра**. Наконец, регистр ESP — это указатель стека.

Следующая группа регистров от CS до GS — сегментные регистры. Это «электронные трилобиты» — атавизмы, оставшиеся от процессора 8088, которому через 16-разрядные адреса было доступно 2^{20} байт памяти. Достаточно сказать, что когда Core i7 работает в режиме использования единого линейного 32-разрядного адресного пространства, их можно смело игнорировать. Регистр EIP (Extended Instruction Pointer — расширенный указатель команд) представляет собой счетчик команд. Регистр EFLAGS — флаговый.

Общий обзор уровня архитектуры набора команд OMAP4430

Архитектура ARM была впервые введена в 1985 году компанией Acorn Computers. Она была основана на исследовании, проведенном в Беркли в 80-е годы [Patterson, 1985; Patterson and Serquin, 1982]. Исходная архитектура ARM (она называлась ARM2) была 32-разрядной и поддерживала 26-разрядное адресное пространство. OMAP4430 использует микроархитектуру ARM Cortex A9, реализующую версию 7 архитектуры ARM, и именно ее мы будем описывать в этой главе. В целях согласованности с остальными частями книги мы будем называть данную систему OMAP4430, хотя на уровне архитектуры набора команд все машины, основанные на ARM Cortex A9, идентичны.

Структура памяти машины OMAP4430 очень проста — линейный массив размером 2^{32} байт. Поддерживаются оба типа нумерации байтов (прямая и обратная); тип нумерации задается в системном блоке памяти, содержимое которого читается сразу же после сброса процессора. Чтобы блок памяти был прочитан правильно, он должен иметь прямой (little-endian) порядок байтов, даже если машина настроена на работу в режиме обратного (big-endian) порядка.

Важно, что предельное число адресуемых байтов больше, чем требуется для реализации архитектуры команд, поскольку в будущем, скорее всего, понадобится увеличить объем памяти, к которой может обращаться процессор. 32-разрядное адресное пространство ARM уже сейчас создает проблемы для разработчиков,

потому что многие системы на базе ARM (например, смартфоны) имеют более 2^{32} байтов памяти. Сейчас для решения этой проблемы проектировщики реализуют значительную часть памяти в виде флэш-накопителя, для работы с которым применяется дисковый интерфейс с поддержкой блочно-ориентированного адресного пространства большего объема. Для устранения этого ограничения (теоретически способного повредить продажам) компания ARM недавно опубликовала определение архитектуры набора команд ARM версии 8 с поддержкой 64-разрядных адресных пространств.

Одна из самых серьезных проблем состоит в том, что архитектура команд ограничивает размер адресуемой памяти. В компьютерных технологиях единственная проблема, у которой не существует обходного решения, — это нехватка битов. Когда-нибудь наши внуки будут удивляться, как же могли работать компьютеры, содержащие всего 32-разрядные адреса и 4 Гбайт памяти, когда средней игре для запуска требуется не менее терабайта.

Архитектура команд ARM достаточно проста, хотя организация регистров была немного усложнена, чтобы сделать вызовы процедур более эффективными. В этой архитектуре программный счетчик отображается на регистровый файл (как регистр R15), так как это позволяет создавать переходы с операциями АЛУ, у которых приемным регистром является R15. Практика показывает, что такая организация регистров создает больше проблем, чем приносит пользы, однако требования обратной совместимости не позволяют от нее отказаться.

В архитектуре ARM присутствуют две группы регистров: 16 32-разрядных регистров общего назначения и 32 32-разрядных регистра для команд с плавающей точкой (если поддерживается сопроцессор VFP). Регистры общего назначения называются R0–R15, но в определенных контекстах используются другие названия. Варианты названий регистров и их функции приведены в табл. 5.1.

Таблица 5.1. Регистры общего назначения в ARM версии 7

Регистр	Другой вариант названия	Назначение
R0–R3	A1–A4	Содержат параметры вызываемой процедуры
R4–R11	V1–V8	Содержат локальные переменные для текущей процедуры
R12	IP	Регистр внутрипроцедурных вызовов (для 32-разрядных вызовов)
R13	SP	Указатель стека
R14	LR	Регистр связи (адрес возврата для текущей функции)
R15	PC	Счетчик команд

Все регистры общего назначения 32-разрядные; все они могут считываться и записываться различными командами загрузки и сохранения. Назначение этих регистров, приведенное в табл. 5.1, отчасти определено по соглашению, отчасти зависит от используемого аппаратного обеспечения. Однако, вообще говоря, не стоит отклоняться от указанного назначения, если вы не являетесь крупным специалистом по компьютерам ARM. Программист должен быть уверен, что программа правильно обращается к регистрам и выполняет с ними допустимые арифметические действия. Например, очень легко загрузить числа с плавающей

точкой в регистры общего назначения, а затем произвести над ними целочисленное сложение — операцию, результатом которой будет полнейшая чепуха, но которую центральный процессор обязательно выполнит, если того потребует программа.

Регистры Vx используются для хранения констант, переменных и указателей, которые нужны во всех процедурах, хотя при необходимости они могут загружаться и перезагружаться при входе в процедуру и при выходе из процедуры. Регистры Ax используются для передачи параметров процедурам, чтобы избежать обращений к памяти. Далее мы расскажем, как это происходит.

Четыре специальных регистра используются для особых целей. Регистр IP позволяет обойти ограничения команды вызова функции ARM (BL), которая не может адресовать все 2^{32} байта адресного пространства. Если целевой адрес находится слишком далеко от команды, то команда вызывает специальный фрагмент кода, передающий управление по адресу в регистре IP. Регистр SP указывает на текущую вершину стека и изменяется, когда слова помещаются в стек или выталкиваются оттуда. Третий специальный регистр — LR — содержит адрес возврата для текущей процедуры. Четвертый специальный регистр — PC — уже упоминался ранее; при сохранении значения в этом регистре выборка команд перенаправляется на указанный адрес. Еще один важный регистр архитектуры ARM — регистр состояния программы, или PSR (Program Status Register) — содержит информацию состояния всех предшествующих вычислений АЛУ. В частности, среди его битов присутствуют признаки нуля, отрицательного значения и переполнения.

В архитектуре ARM (с сопроцессором VFP) есть также 32 32-разрядных регистра. К этим регистрам можно обращаться либо как к 32 вещественным значениям одинарной точности, либо как к 16 64-разрядным вещественным значениям двойной точности. Размер регистра определяется командой; практически все команды с плавающей точкой в ARM существуют в разновидностях с одинарной и двойной точностью.

Архитектура ARM относится к категории **архитектур загрузки и сохранения**. Это означает, что единственные команды, которые непосредственно обращаются к памяти, — это `LOAD` (загрузка) и `STORE` (сохранение), обеспечивающие перемещение данных между регистрами и памятью. Все операнды для команд арифметических и логических операций должны извлекаться из регистров или предоставляться самой командой (без обращений к памяти), а все результаты должны сохраняться в регистрах (но не в памяти).

Обзор уровня архитектуры набора команд ATmega168

В качестве третьего примера мы рассмотрим микросхему ATmega168. В отличие от процессоров Core i7 (которые в основном используются в настольных компьютерах и серверных фермах) и OMAP4430 (которые устанавливаются в основном в телефонах, планшетных компьютерах и других мобильных устройствах), микросхемы ATmega168 обычно используются в низкопроизводительных встроенных системах (от уличных светофоров до будильников) для обработки сигналов от кнопок, световых индикаторов и других элементов пользовательского интерфейса. В данном подразделе представлена краткая техническая характеристика ATmega168 и родственных микросхем.

ATmega168 работает всего в одном режиме и не имеет аппаратных средств защиты. Они и не нужны — ведь 8051 никогда не исполняет одновременно несколько программ, одна из которых может быть вредоносной. Модель памяти чрезвычайно проста. Имеется 16 Кбайт программной памяти и 1 Кбайт памяти данных. Эти пространства разделены, так что адрес может относиться к разным блокам памяти в зависимости от того, к чему относится обращение — к области данных или области программ. Разделение пространств памяти делает возможной реализацию пространства программ в ПЗУ, а пространства данных — в ОЗУ.

Возможно несколько вариантов реализации памяти в зависимости от того, какую сумму проектировщик согласен заплатить за процессор. В простейшем варианте (ATmega48) предусматривается 4 Кбайт флэш-памяти для программ и 512 байт ОЗУ для данных. Как флэш-память, так и ОЗУ размещаются в одном корпусе с микросхемой. С учетом области применения такого объема памяти обычно вполне достаточно, а совместное размещение процессора и двух модулей памяти считается серьезным достоинством. В модели ATmega88 емкость обоих модулей увеличена вдвое: 8 Кбайт ПЗУ и 1 Кбайт ОЗУ.

ATmega168 использует двухуровневую организацию памяти для улучшения защиты программ. ПЗУ программ делится на *секцию загрузчика* и *секцию приложений*; размеры секций определяются битами, программируемыми всего один раз в момент первой подачи питания на микроконтроллер. По соображениям безопасности флэш-память может обновляться только кодом из секции загрузчика. Благодаря этой особенности код, помещенный в секцию приложений (включая загруженные сторонние приложения), можно запускать в полной уверенности, что он не помешает работе другого кода в системе (потому что этот код будет выполняться в пространстве приложений, из которого запись во флэш-память невозможна). Чтобы полностью исключить любые эксцессы, поставщик может снабдить код электронной подписью. В этом случае загрузчик загрузит код во флэш-память только при наличии подписи утвержденного поставщика, то есть код будет выполняться в системе только с «благословения» надежного источника. Решение получается достаточно гибким: оно позволяет заменить даже загрузчик, если новый код будет иметь правильную электронную подпись. Apple и TiVo используют похожий способ для защиты кода, выполняемого на их устройствах.

ATmega168 содержит 32 8-разрядных регистра общего назначения. Для обращения к ним команды используют 5-разрядное поле, определяющее используемый регистр. Регистры обозначены именами с R0 до R31. У регистров ATmega168 имеется одна особенность: они также присутствуют в пространстве памяти. Байт 0 пространства данных эквивалентен регистру R0 из набора 0. Когда команда изменяет R0, а потом читает байт памяти 0, она находит в нем новое значение R0. Аналогичным образом байт 1 памяти соответствует R1 и т. д., до байта R31. Схема отображения представлена на рис. 5.4.

Непосредственно над 32 регистрами общего назначения, по адресам памяти 32–95, расположены 64 байта памяти, зарезервированной для регистров устройств ввода-вывода, включая внутренние устройства однокристальной системы.

Помимо 4 наборов из 8 регистров, в ATmega168 есть несколько специальных регистров, самые важные из которых изображены на рис. 5.4. В *регистре состояния* содержатся (слева направо): бит разрешения прерывания, хранение копи-

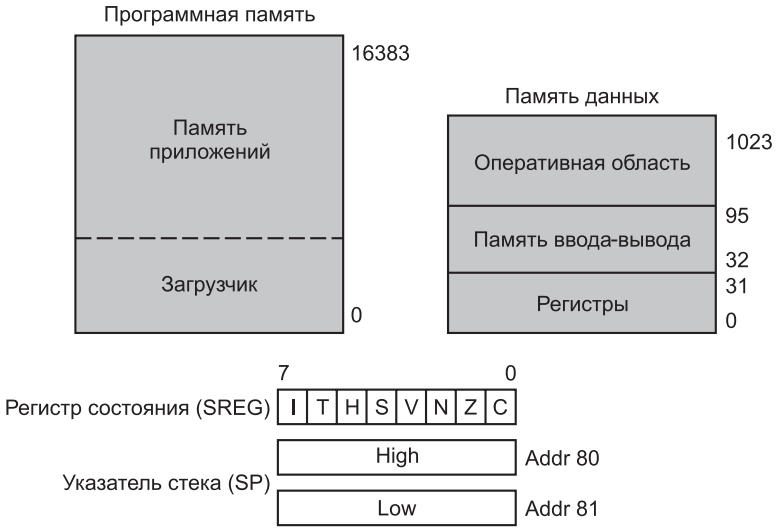


Рис. 5.4. Организация регистров и памяти у ATmega168

руемого бита, бит полупереноса, бит знака, бит переполнения, флаг отрицательности, флаг нуля и бит переноса. Значения всех этих битов, за исключением бита разрешения прерывания, вычисляются в результате арифметических операций.

Бит I регистра состояния предоставляет возможность глобального включения/отключения прерываний. Если бит I равен 0, все прерывания отключены. Сброс этого бита позволяет отключать последующие прерывания одной командой. Установка этого бита разрешает любые незавершенные прерывания, а также прерывания, которые поступят в будущем. Каждое устройство связывается с битом разрешения прерываний. Если бит устройства установлен вместе с битом глобального разрешения прерываний I, устройство может прерывать работу процессора.

Указатель стека SP содержит текущий адрес памяти данных, по которому команды PUSH и POP будут обращаться к своим данным (по аналогии с одноименной командой виртуальной машины Java из главы 4). Указатель стека находится в памяти ввода-вывода по адресу 80. Одного 9-разрядного байта памяти недостаточно для адресации 1024 байт памяти данных, поэтому указатель стека образуется двумя смежными ячейками памяти.

Типы данных

Всем компьютерам нужны данные. Для многих компьютерных систем основной задачей является обработка финансовых, промышленных, научных, технических и других данных. Внутри компьютера данные должны быть представлены в какой-либо особой форме. На уровне архитектуры набора команд используются различные типы данных. Они описаны в этом разделе.

Ключевым вопросом является наличие аппаратной поддержки того или иного типа данных. Под аппаратной поддержкой подразумевается, что одна или несколько команд ожидают получать данные в определенном формате, и пользо-

ватель не может задействовать другой формат. Например, бухгалтеры привыкли писать знак «минус» у отрицательных чисел справа, а специалисты по вычислительной технике — слева. Предположим, что, пытаясь произвести впечатление на своего начальника, глава компьютерного центра в бухгалтерской фирме изменил все числа во всех компьютерах, чтобы знаковый бит был самым правым (а не самым левым). Несомненно, это произведет большое впечатление на начальника, поскольку все программное обеспечение перестанет работать. Аппаратное обеспечение требует определенного формата для целых чисел и перестанет работать должным образом, если целые числа поступят в другом формате.

Теперь рассмотрим другую бухгалтерскую фирму, только что заключившую договор на проверку федерального долга (размера задолженности правительства США всем контрагентам). 32-разрядная арифметика здесь не подойдет, поскольку числа превышают значение 2^{32} (около 4 миллиардов). Одно из возможных решений — использовать два 32-разрядных целых числа для представления каждого числа, то есть все 64 бита. Если машина не поддерживает такие **числа удвоенной точности**, все арифметические операции над ними должны выполняться программно, то есть эти две части могут располагаться в памяти в произвольном порядке, поскольку для аппаратного обеспечения это не важно. Это — пример типа данных без аппаратной поддержки и, следовательно, без аппаратной реализации. В следующих подразделах мы рассмотрим типы данных, которые поддерживаются аппаратно и для которых требуются специальные форматы.

Числовые типы данных

Типы данных можно разделить на две категории: числовые и нечисловые. Среди числовых типов данных главными являются целые числа. Они бывают различной длины: обычно 8, 16, 32 и 64 бита. Целые числа применяются для подсчета различных предметов (например, позволяют узнать, сколько на складе отверток), для идентификации различных объектов (например, банковских счетов), а также для других целей. В большинстве современных компьютеров целые числа хранятся в двоичном виде, хотя в прошлом использовались и другие системы. Двоичные числа обсуждаются в приложении А.

Некоторые компьютеры поддерживают целые числа и со знаком, и без знака. В целом числе без знака нет знакового бита, и все биты содержат данные. Этот тип данных имеет преимущество: у него есть дополнительный бит, поэтому 32-разрядное слово может содержать целое число без знака от 0 до $2^{32}–1$ включительно. Двоичное целое число со знаком, напротив, может содержать числа только до $2^{31}–1$, но зато может принимать и отрицательные значения.

Для выражения нецелых чисел (например, 3,5) используются числа с плавающей точкой. О них рассказывается в приложении Б. Их длина составляет 32, 64, а иногда и 128 бит. В большинстве компьютеров есть команды для выполнения операций с числами с плавающей точкой. Во многих компьютерах имеются отдельные регистры для целочисленных операндов и для операндов с плавающей точкой.

Некоторые языки программирования (в частности COBOL) поддерживают десятичное представление чисел. Машины, предназначенные для программ на языке COBOL, часто поддерживают десятичные числа аппаратно — для этого

десятичный разряд кодируется четырьмя битами, а затем два десятичных разряда объединяются в байт (двоично-десятичный формат). Однако результаты арифметических действий над такими десятичными числами будут некорректны, поэтому требуются специальные команды для коррекции десятичной арифметики. Эти команды должны контролировать перенос бита 3. Вот почему код условия часто содержит бит служебного переноса. Между прочим, проблема 2000 года возникла из-за программистов на языке COBOL, которые решили, что год экономнее представлять в виде двух десятичных разрядов (8 бит), а не в виде 16-разрядного двоичного числа — или хотя бы 8-разрядного двоичного числа, способного представлять больше значений (256), чем в двоично-десятичном упакованном представлении (100). Такая вот «оптимизация»!

Нечисловые типы данных

Хотя самые первые компьютеры работали в основном с числами, современные машины часто используются для исполнения нечисловых приложений, например для обработки текстов или управления базами данных. Для этих приложений нужны другие, нечисловые, типы данных. Они часто поддерживаются командами уровня архитектуры набора команд. Здесь очень важны символы, хотя не каждый компьютер обеспечивает аппаратную поддержку для них. Наиболее распространенными символьными кодами являются ASCII и Unicode. Они поддерживают 7-разрядные и 16-разрядные символы соответственно. Эти коды обсуждались в главе 2.

На уровне архитектуры набора команд часто имеются особые команды, предназначенные для операций со строками. Эти строки иногда разграничиваются специальным символом в конце. Вместо терминального символа для определения конца строки может использоваться поле длины строки. Строковые команды позволяют выполнять копирование, поиск и редактирование строк, а также другие действия.

К нечисловым относится также очень важный логический тип данных, содержащий булевы значения. Этих значений всего два: истина и ложь. Теоретически булево значение можно представлять единственным битом: 0 — ложь, 1 — истина (или наоборот). На практике же используется байт или слово, поскольку отдельные биты в байте не имеют собственных адресов и, следовательно, к ним трудно обращаться. Во многих системах применяется следующее соглашение: 0 означает ложь, а любое другое значение — истину.

Единственная ситуация, в которой булево значение представляется одним битом — это массив битов, поэтому 32-разрядное слово может содержать 32 булевых значения. Такая структура данных называется **битовым отображением**, или **битовой картой**. Битовое отображение встречается в различных контекстах, например, оно позволяет отслеживать свободные блоки на диске — в этом случае каждый бит отображает состояние каждого блока. Если диск содержит n блоков, тогда битовое отображение будет содержать n бит.

Последний тип данных — это указатели, которые представляют собой машинные адреса. Мы уже неоднократно рассматривали указатели. В машинах Mic-x регистры SP, PC, LV и CPP — это примеры указателей. Доступ к переменной на фиксированном расстоянии от указателя (а именно так работает команда `ILOAD`) широко поддерживается всеми машинами. При всей своей полезности указатели

также являются причиной многочисленных ошибок, часто с весьма серьезными последствиями. При работе с ними необходима крайняя осторожность.

Типы данных процессора Core i7

Core i7 поддерживает двоичные целые числа со знаком, целые числа без знака, двоично-десятичный формат и числа с плавающей точкой по стандарту IEEE 754 (табл. 5.2). Происходя от 8/16-разрядных предков, этот процессор работает с целыми числами такой же длины и 32-разрядными данными, а также поддерживает многочисленные арифметические команды, булевы операции и операции сравнения. Кроме того, процессор может работать в 64-разрядном режиме, который также поддерживает 64-разрядные регистры и операции. Операнды не обязательно должны быть выровнены в памяти, но если адреса слов кратны значению 4 байта, процессор работает быстрее.

Таблица 5.2. Числовые типы данных процессора Core i7

Тип	8 бит	16 бит	32 бита	64 бита
Целые числа со знаком	Да	Да	Да	Да (64-разрядные)
Целые числа без знака	Да	Да	Да	Да (64-разрядные)
Двоично-десятичные целые числа	Да			
Числа с плавающей точкой			Да	Да

Core i7 также может манипулировать 8-разрядными ASCII-символами: существуют специальные команды для копирования и поиска строк символов. Эти команды используются и для строк, длина которых известна заранее, и для строк, в конце которых стоит специальный терминальный символ. Строковые команды часто используются в библиотеках для работы со строковыми данными.

Типы данных машины OMAP4430

OMAP4430 поддерживает множество форматов данных (табл. 5.3). Только в целочисленной области эта система может обрабатывать 8-, 16- и 32-разрядные операнды со знаком и без знака. С этими «малыми» типами данных OMAP4430 работает несколько умнее, чем Core i7. Во внутренней реализации OMAP4430 представляет собой 32-разрядную машину с 32-разрядными трактами данных и командами. Для команд загрузки и сохранения программа может задать размер и знак загружаемых значений (например, команда загрузки байта со знаком: LDRSB). Затем значение преобразуется командами загрузки в совместимое 32-разрядное значение. Аналогичным образом команды сохранения также задают размер и знак сохраняемого значения и работают только с указанной частью входных регистров.

Целые числа со знаком представлены в дополнительном коде. Кроме того, имеются операнды с плавающей точкой по 32 и 64 бит, которые соответствуют стандарту IEEE 754 (для 32- и 64-разрядных чисел). Двоично-десятичные числа не поддерживаются. Все операнды должны быть выровнены в памяти. Символьные и строковые типы данных не поддерживаются на уровне команд — все операции с ними осуществляются исключительно на уровне программного обеспечения.

Таблица 5.3. Числовые типы данных компьютера OMAP4430

Тип	8 бит	16 бит	32 бита	64 бита
Целые числа со знаком	Да	Да	Да	
Целые числа без знака	Да	Да	Да	
Двоично-десятичные целые числа				
Числа с плавающей точкой			Да	Да

Типы данных ATmega168

Количество типов данных у ATmega168 строго ограничено. Разрядность всех регистров — а значит, целых чисел и символов — составляет 8 бит. Фактически единственным типом данных для выполнения арифметических операций, который поддерживается аппаратно, является 8-разрядный байт (табл. 5.4).

Таблица 5.4. Числовые типы данных для ATmega168

Тип	8 бит	16 бит	32 бита	64 бита
Целые числа со знаком	Да			
Целые числа без знака	Да	Да		
Двоично-десятичные целые числа				
Числа с плавающей точкой				

Для упрощения работы с памятью ATmega168 также имеет ограниченную поддержку 16-разрядных указателей без знака. 16-разрядные указатели X, Y и Z могут формироваться посредством объединения пар 8-разрядных регистров R26/R27, R28/R29 и R30/R31 соответственно. Когда операция загрузки использует X, Y и Z в качестве адресного операнда, процессор также может дополнительно увеличить или уменьшить значение на 1.

Форматы команд

Команда состоит из кода операции и некоторой дополнительной информации, позволяющей узнать, например, откуда поступают операнды и куда должны отправляться результаты. Процесс определения, где находятся операнды (то есть их адреса), называется **адресацией**.

На рис. 5.5 показано несколько возможных форматов команд уровня архитектуры набора команд. Команды всегда содержат код операции. В команде могут присутствовать ни одного, один, два или три адреса.

В одних машинах все команды по длине одинаковы, в других могут быть разными. Кроме того, команды по длине могут быть короче слова, длиннее слова или быть равными слову. Если все команды одной длины, то это упрощает декодирование, но часто требует излишнего пространства, поскольку все команды должны быть такой же длины, как самая длинная. На рис. 5.6 показано несколько возможных соотношений между размерами команды и слова.

Рис. 5.5. Четыре формата команд: безадресная команда (*а*); одноадресная команда (*б*); двухадресная команда (*в*); трехадресная команда (*г*)

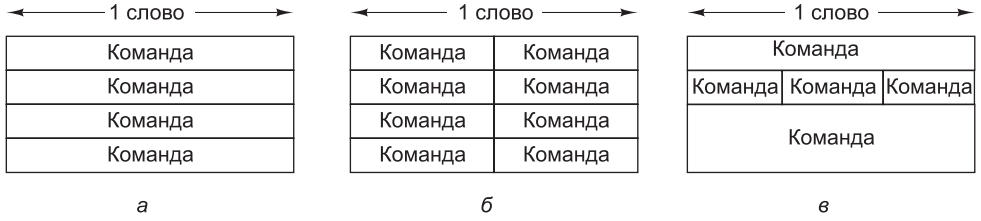


Рис. 5.6. Некоторые возможные отношения между размерами команды и слова

Критерии проектирования форматов команд

Выбирая форматы команд для своей машины, проектировщик должен принять во внимание ряд факторов. Не стоит недооценивать сложность этого решения. Если компьютер с коммерческой точки зрения получается удачным, набор команд может существовать на протяжении 40 лет и более. Имеет огромное значение возможность на протяжении некоторого времени добавлять новые команды и другие элементы, но только в том случае, если архитектура (и компания, создавшая эту архитектуру) протянет достаточно долго.

Эффективность конкретной архитектуры команд зависит от технологии, которая применялась при разработке компьютера. За длительный период времени эта технология значительно изменится, и некоторые характеристики архитектуры команд окажутся (если оглянуться лет на 20 назад) неудачными. Например, если доступ к памяти осуществляется быстро, то подойдет стековая архитектура (как в IJVM), но если доступ к памяти медленный, тогда желательно иметь побольше регистров (как в OMAP4430). Тем читателям, которые считают, что выбор сделать просто, мы предлагаем взять лист бумаги и записать следующие предположения:

- Какова будет типичная частота тактового генератора через 20 лет?
- Каково будет типичное время доступа к ОЗУ через 20 лет?

Аккуратно сложите этот лист бумаги и спрячьте его в надежном месте, а через 20 лет разверните и прочитайте, что на нем написано. Наиболее самоуверенные могут выложить свои пророчества в Интернете.

Конечно, даже дальновидные разработчики не всегда могут сделать правильный выбор. А если бы и смогли, то проработали бы недолго — если предлагаемая ими архитектура команд окажется дороже, чем у конкурентов, компания долго не продержится, и мир не оценит элегантности принятых решений.

При прочих равных факторах короткие команды лучше длинных. Программа, состоящая из n 16-разрядных команд, занимает в два раза меньше пространства памяти, чем программа из n 32-разрядных. Поскольку цены на память постоянно падают, этот фактор может утратить значение в будущем, но, к сожалению, программное обеспечение разрастается гораздо быстрее, чем снижаются цены.

Кроме того, минимизация размера команд может усложнить их декодирование и параллельное выполнение. Следовательно, стремление уменьшить размер команд должно уравновешиваться стремлением сократить время их декодирования и выполнения.

Есть еще одна очень важная причина минимизации длины команд, и она становится все важнее с увеличением скорости работы процессоров: пропускная способность памяти (число битов в секунду, которое память может предоставлять). Значительное повышение быстродействия процессоров за последнее десятилетие не соответствует увеличению пропускной способности памяти. Ограничения здесь связаны с неспособностью системы памяти передавать команды и операнды с той же скоростью, с которой процессор может их обрабатывать. Пропускная способность памяти зависит от технологии разработки. Встречаемые на этом пути трудности имеют отношение не только к основной памяти, но и ко всем видам кэш-памяти.

Если пропускная способность кэш-памяти команд составляет t бит/с, а средняя длина команды — r бит, то кэш-память способна передавать самое большое t/r команд в секунду. Отметим, что это — *верхний предел* скорости, с которой процессор может выполнять команды, хотя в настоящее время предпринимаются попытки преодолеть данный барьер. Ясно, что скорость, с которой могут выполняться команды (то есть быстродействие процессора), может ограничиваться длиной команд. Чем короче команды, тем быстрее работает процессор. Поскольку современные процессоры способны выполнять несколько команд за один цикл, то вызов нескольких команд за цикл обязателен. Этот аспект применения кэш-памяти команд делает размер команд важным критерием, который нужно учитывать при разработке.

Еще один критерий — достаточный объем пространства в формате команды для представления всех требуемых операндов. Машина, поддерживающая 2^n операций, не может иметь длину команды менее n бит. В этом случае в коде операции было бы недостаточно места для того, чтобы указать, какая нужна команда. К тому же история снова и снова доказывает, как неразумно поступали проектировщики, не оставившие свободных кодов операций для будущих дополнений набора команд.

Третий критерий связан с числом битов в адресном поле. Рассмотрим проект машины с 8-разрядными символами и основной памятью, которая должна содержать 2^{32} символов. Разработчики могут по своему усмотрению связать последовательные адреса с блоками по 8, 16, 24 или 32 бита.

Представим, что бы случилось, если бы команда разработчиков разбилась на две воюющие группы, одна из которых утверждает, что основной единицей

памяти должен быть 8-разрядный байт, а другая требует, чтобы основной единицей памяти было 32-разрядное слово. Первая группа предложила бы память из 2^{32} байт с номерами 0, 1, 2, 3, ..., 4 294 967 295. Вторая группа предложила бы память из 2^{30} слов с номерами 0, 1, 2, 3, ..., 1 073 741 823.

Первая группа скажет, что для того, чтобы сравнить два символа при организации по 32-разрядным словам, программе приходится не только вызывать из памяти слова, содержащие эти символы, но и выделять соответствующий символ из каждого слова для сравнения. А это потребует дополнительных команд и, следовательно, дополнительного пространства. 8-разрядная организация, напротив, обеспечивает адресацию каждого символа, что значительно упрощает процедуру сравнения.

Сторонники 32-разрядной организации скажут, что их проект требует всего лишь 2^{30} отдельных адресов, что дает длину адреса всего 30 бит, тогда как при 8-разрядной организации требуется целых 32 бита для обращения к той же самой памяти. Если адрес короткий, то и команда будет более короткой. Она займет меньше пространства в памяти и к тому же для ее вызова потребуется меньше времени. Кроме того, 32-разрядные адреса могут использоваться для обращения к памяти в 16 Гбайт вместо каких-то там 4 Гбайт.

Этот пример демонстрирует, что для получения оптимальной дискретности памяти требуются более длинные адреса и, следовательно, более длинные команды. Одна крайность — это организация памяти, при которой адресуется каждый бит (например, Burroughs B1700). Другая крайность — это память, состоящая из очень длинных слов (например, серия CDC Cyber содержала 60-разрядные слова).

Современные компьютерные системы пришли к компромиссу, который, в каком-то смысле, объединил в себе худшие качества обоих вариантов. Они требуют, чтобы адреса были у отдельных байтов, но при обращении к памяти всегда считываются одно, два, а иногда даже четыре слова сразу. Например, в результате считывания одного байта из памяти на машине Core i7 единовременно вызываются минимум 8 байт, а иногда и вся строка кэш-памяти размером 64 байта.

Расширение кода операций

В предыдущем подразделе мы увидели, что короткие адреса и хорошая дискретность памяти могут противоречить друг другу. В этом разделе мы рассмотрим компромиссы, связанные с кодами операций и адресами. Рассмотрим команду размером $n + k$ бит с кодом операции в k бит и одним адресом в n бит. Такая команда допускает 2^k различных операций и 2^n адресуемых ячеек памяти. В качестве альтернативы те же $n + k$ бит можно разбить на код операции в $k - 1$ бит и адрес в $n + 1$ бит. При этом будет либо в два раза меньше команд, но в два раза больше памяти, либо то же количество памяти, но дискретность вдвое выше. Код операции в $k + 1$ бит и адрес в $n - 1$ бит дает большее количество операций, но за это придется расплачиваться либо сокращением количества ячеек памяти, либо не очень удачной дискретностью при том же объеме памяти. Наряду с подобными простыми компромиссами между битами кода операции и битами адреса существуют и более сложные. Обсуждаемый здесь механизм называется **расширением кода операций**.

Понятие расширения кода операций можно пояснить на примере. Рассмотрим машину, в которой длина команд составляет 16 бит, а длина адресов — 4 бита, как показано на рис. 5.7. Это вполне разумно для машины, содержащей 16 регистров (а следовательно, 4-разрядный адрес регистра), с которыми совершаются все арифметические операции. Один из возможных вариантов — включение в каждую команду 4-разрядного кода операции и трех адресов, что дает 16 трехадресных команд.

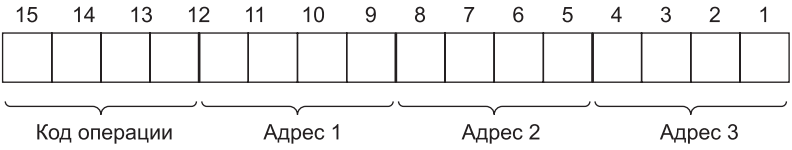


Рис. 5.7. Команда с 4-разрядным кодом операции и тремя 4-разрядными адресными полями

Если разработчикам нужно 15 трехадресных команд, 14 двухадресных команд, 31 одноадресная команда и 16 безадресных команд, они могут использовать коды операций от 0 до 14 в качестве трехадресных команд, а код операции 15 уже интерпретировать по-другому (рис. 5.8).

Это значит, что код операции 15 содержится в битах с 8 по 15, а не с 12 по15. Биты с 0 по 3 и с 4 по 7, как и раньше, формируют два адреса. Все 14 двухадресных команд содержат число 1111 в старших четырех битах и числа от 0000 до 1101 в битах с 8 по 11. Команды с числом 1111 в старших четырех битах и числом 1110 или 1111 в битах с 8 по 11 рассматриваются особо. Они интерпретируются так, как будто их коды операций находятся в битах с 4 по 15. В результате получаем 32 новых кода операций. А поскольку требуется всего 31 код, то код 111111111111 означает, что действительный код операции находится в битах с 0 по 15, что дает 16 безадресных команд.

Как видим, код операции становится все длиннее и длиннее: трехадресные команды имеют 4-разрядный код операции, двухадресные команды — 8-разрядный, одноадресные команды — 12-разрядный, а безадресные команды — 16-разрядный.

Идея расширения кода операций наглядно демонстрирует компромисс между пространством для кодов операций и пространством для другой информации. Однако на практике все не так просто и понятно, как в нашем примере. Есть только два способа изменения размера кода операций. С одной стороны, можно иметь все команды одинаковой длины, приписывая самые короткие коды операций тем командам, которым нужно больше всего битов для спецификации чего-либо другого. С другой стороны, можно свести к минимуму *средний* размер команды, если выбрать самые короткие коды операций для часто используемых команд и самые длинные — для редко используемых.

Если довести эту идею до конца, можно свести к минимуму среднюю длину команды, закодировав каждую команду, чтобы максимально уменьшить число требуемых битов. К сожалению, это ведет к наличию команд разных размеров, причем не выровненных в границах байтов. Пока существуют архитектуры команд с таким свойством (например, Intel 432), выравнивание будет иметь большое значение для быстрого декодирования команд. Тем не менее эта идея

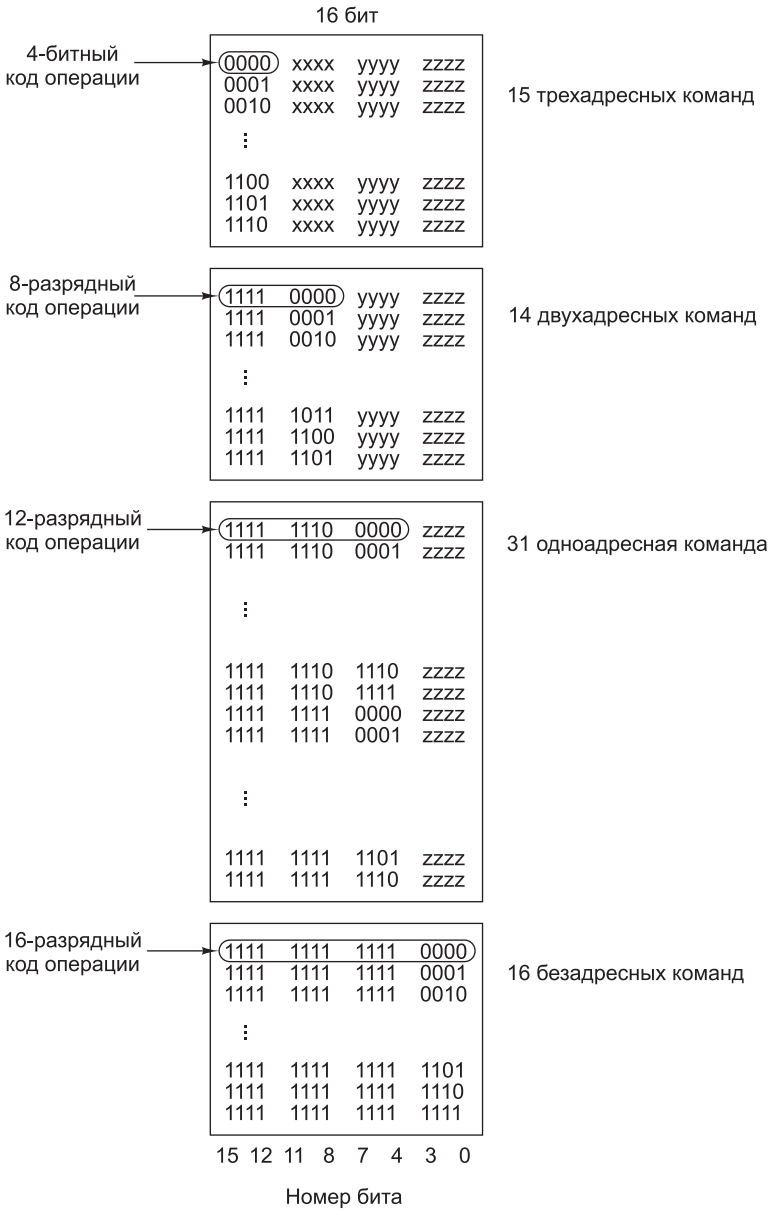


Рис. 5.8. Расширение кода операции обеспечивает 15 трехадресных команд, 14 двухадресных команд, 31 одноадресную команду и 16 безадресных команд. Поля xxxx, yyyy и zzzz — это 4-разрядные адресные поля

часто применяется на уровне байтов. Далее мы рассмотрим архитектуру JVM-команд, чтобы показать, как можно менять форматы команд, чтобы максимально уменьшить размер программы.

Форматы команд процессора Core i7

Форматы команд процессора Core i7 очень сложны и нерегулярны. Они содержат до шести полей разной длины, пять из которых не обязательны (общая схема показана на рис. 5.9). Такая ситуация сложилась из-за того, что архитектура развивалась на протяжении нескольких поколений и ранее в нее были включены не очень удачные варианты команд. Из-за требования обратной совместимости позднее их нельзя было изменить. Например, если один из операндов команды находится в памяти, то другой не может там находиться. В результате существуют команды сложения двух регистров, команды прибавления регистра к слову памяти, команды прибавления слова памяти к регистру, но не существует команд для прибавления одного слова памяти к другому слову памяти.

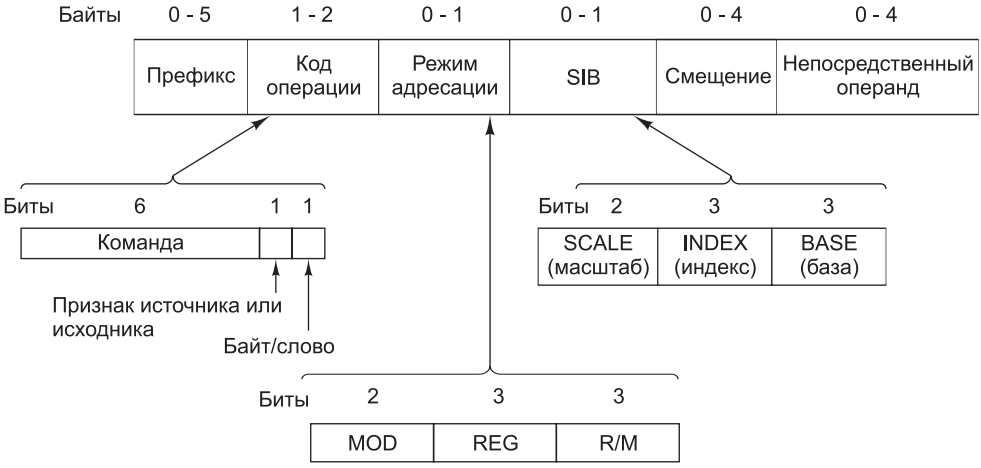


Рис. 5.9. Форматы команд процессора Core i7

В первых архитектурах Intel все коды операций были размером 1 байт, хотя для изменения некоторых команд часто использовался так называемый префиксный байт. **Префиксный байт** — это дополнительный код операции, который ставится перед командой, чтобы изменить ее действие. Примером применения префиксного байта может служить команда `WIDE` в машинах IJVM. К сожалению, в какой-то момент компания Intel исчерпала запасы кодов операций, и один код операции, 0xFF, стал **служебным кодом**, указывающим на наличие второго байта команды.

Отдельные биты кодов операций процессора Core i7 дают довольно мало информации о команде. Единственной структурой такого рода в поле кода операции является младший бит в некоторых командах, который указывает, что именно вызывается — слово или байт, а также соседний бит, который указывает, является ли адрес памяти (если он вообще есть) источником или приемником. Таким образом, в большинстве случаев код операции нужно полностью декодировать, чтобы установить, к какому классу относится выполняемая операция и, следовательно, какова длина команды. Это значительно снижает производительность, поскольку декодирование необходимо производить даже до определения начала следующей команды.

В большинстве команд вслед за байтом кода операции, который указывает местонахождение операнда в памяти, следует второй байт, несущий всю информацию об операнде. Эти 8 бит распределены по 2-разрядному полю MOD и двум 3-разрядным регистровым полям REG и R/M. Иногда первые 3 бита этого байта используются в качестве расширения для кода операции, давая в сумме 11 бит для кода операции. Тем не менее 2-разрядное поле означает, что существуют только 4 способа обращения к операндам и один из операндов всегда должен быть регистром. Логически должен идентифицироваться любой из регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, но правила кодирования команд запрещают некоторые комбинации, которые отводятся для особых случаев. В некоторых типах команд требуется дополнительный байт, называемый **SIB** (Scale, Index, Base — **масштаб**, **индекс**, **база**), который дает дополнительную спецификацию. Эта схема не идеальна, она является компромиссом между требованием обратной совместимости и желанием учесть новые особенности, которые не были предусмотрены изначально.

Добавим еще, что некоторые команды имеют 1, 2 или 4 дополнительных байта для определения адреса команды (смещение), а иногда еще 1, 2 или 4 байта, содержащих константу (непосредственный операнд).

Форматы команд процессора OMAP4430

Архитектура команд процессора OMAP4430 состоит из 16- и 32-разрядных команд, выровненных в памяти. Команды очень просты. Каждая из них выполняет только одно действие. Типичная команда задает два регистра, в которых находятся исходные операнды, и один выходной регистр. 16-разрядные команды представляют собой упрощенные версии 32-разрядных команд. Они выполняют те же операции, но допускают использование только двух регистровых операндов (то есть приемный регистр должен совпадать с одним из источников), и только первые 8 регистров могут задаваться в качестве входных. Проектировщики архитектуры ARM присвоили этой упрощенной версии набора команд ARM обозначение «Thumb».

Вместо одного из регистров команда может использовать 3-, 8-, 12-, 16- или 24-разрядную константу без знака. При выполнении команды `LOAD` два регистра (или один регистр и 8-разрядная константа со знаком) складываются для определения считываемого адреса памяти. Данные оттуда записываются в другой указанный в команде регистр.

Формат 32-разрядных команд ARM показан на рис. 5.10. Внимательный читатель заметит, что некоторые форматы состоят из одинаковых полей (например, LONG MULTIPLY и SWAP). В случае команды SWAP блок декодирования знает, что имеет дело с командой SWAP только в том случае, если видит, что комбинация значений полей для MUL недопустима. Для расширений и архитектуры Thumb были добавлены дополнительные форматы. На момент написания книги существовал 21 формат команды, причем их количество увеличивалось. (Долго ли осталось ждать того дня, когда появится реклама «самого сложного в мире RISC-процессора»?) Впрочем, большинство команд продолжает использовать форматы, показанные на рисунке.

31 2827 1615 87 0

															Тип команды
Cond	0 0	I	Код операции	S	Rn	Rd	Операнд2								Data processing / PSR Transfer
Cond	0 0 0 0 0 0	A	S	Rd	Rn	RS	1 0 0 1	Rm							Multiply
Cond	0 0 0 0 1	U	A	S	RdHi	RdLo	RS	1 0 0 1	Rm						Long Multiply
Cond	0 0 0 1 0	B	0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm							Swap
Cond	0 1	I	P	U	B	W	L	Rn	Rd	Смещение					Load/Store Byte/Word
Cond	1 0 0	P	U	S	W	L	Rn	Список регистров							Load/Store Multiple
Cond	0 0 0	P	U	1	W	L	Rn	Rd	Смещение 1	1	S	H	1	Смещение 2	Halfword transfer: Immediate offset
Cond	0 0 0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	Halfword transfer: Register offset
Cond	1 0 1	L	Смещение												Branch
Cond	0 0 0 1	0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn								Branch Exchange
Cond	1 1 0	P	U	N	W	L	Rn	CRd	CPNum	Смещение					Coprocessor data transfer
Cond	1 1 1 0	Op1	CRn	CRd	CPNum	Op2	0	CRm							Coprocessor data operation
Cond	1 1 1 0	Op1	L	CRn	Rd	CPNum	Op2	1	CRm						Coprocessor register transfer
Cond	1 1 1 1	Номер прерывания													Software interrupt

Рис. 5.10. Форматы 32-разрядных команд ARM

Биты 26 и 27 каждой команды помогают определить формат команды и сообщают аппаратному обеспечению, где найти оставшуюся часть кода операции, если она есть. Например, если биты 26 и 27 равны нулю, а бит 25 также равен нулю (операнд не является непосредственным), а сдвиг входного операнда не является недопустимым (признак команды MULTIPLY или BRANCH EXCHANGE), то оба источника являются регистрами. Если бит 25 равен 1, то один источник является регистром, а другой — константой в диапазоне от 0 до 4095. В обоих случаях местом сохранения результатов всегда является регистр. Достаточный объем пространства декодирования предоставляется для 16 команд, все из которых используются в настоящее время.

Поскольку все команды 32-разрядные, включить в команду 32-разрядную константу невозможно. Команда `MOVT` устанавливает 16 старших битов 32-разрядного регистра, оставляя пространство для другой команды, чтобы установить оставшиеся 16 бит. Это единственная команда такого необычного формата.

У каждой 32-разрядной команды в старших битах (с 28 по 31) содержится одно и то же 4-разрядное поле. Это поле условия, с которым любая команда становится **предикатной**. Предикатная команда выполняется процессором как обычно, но перед записью результата в регистр (или в память) она сначала проверяет условие команды. Для команд ARM условие базируется на состоянии регистра состояния процессора (PSR), содержащего свойства последней арифметической операции (нуль, отрицательное значение, переполнение и т. д.). Если условие не выполнено, результат условной команды игнорируется.

В формате команды перехода кодируется самое большое непосредственное значение, используемое для вычисления целевого адреса переходов и вызовов функций. Эта команда отличается от других: только в ней для определения адреса необходимы 24 бита данных. В этой команде используется один 3-разрядный код операции. Адрес представляет собой целевой адрес, разделенный на четыре. Таким образом, относительно текущей команды диапазон составляет примерно $\pm 2^{25}$ байт.

Разумеется, проектировщики набора команд ARM хотели использовать для определения команд все возможные комбинации битов, включая недопустимые (в остальных случаях) комбинации операндов. Такой подход усложняет логику декодирования, но в то же время позволяет закодировать максимальное количество операций в 16- или 32-разрядной команде фиксированной длины.

Форматы команд ATmega168

В ATmega168 предусмотрено шесть простых форматов команд (рис. 5.11). Размер команд может быть равен двум или четырем байтам. Формат 1 состоит из кода операции и двух регистровых операндов, оба из которых являются входными, а в один помещается результат команды. Например, команда ADD для регистров использует именно этот формат.

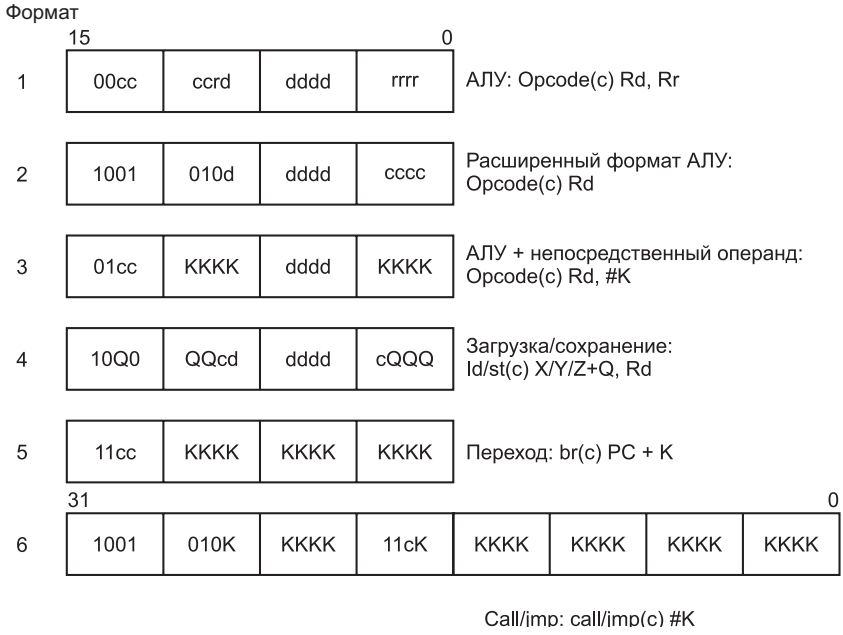


Рис. 5.11. Форматы команд ATmega168 AVR

Формат 2 также состоит из 16 бит, с 16 дополнительными кодами операций и 5-разрядным номером регистра. Этот формат увеличивает количество операций, закодированных в наборе команд, за счет сокращения количества операндов до 1. Команды, использующие этот формат, выполняют унарную операцию — они получают единственное входное значение в регистре и записывают результат операции в тот же регистр. Например, к этому типу относятся команды изменения знака и инкремента.

Команды формата 3 имеют один 8-разрядный непосредственный операнд без знака. Чтобы такое большое непосредственное значение поместилось в 16-разрядной команде, команда может иметь всего один регистровый операнд (используемый для ввода и вывода), причем регистр может находиться только в диапазоне R16–R31 (ограничивающем кодирование операнда четырьмя разрядами). Кроме

того, количество битов кода операции уменьшено вдвое, вследствие чего только четыре команды могут использовать этот формат (SUBCI, SUBI, ORI и ANDI).

Формат 4 предназначен для команд загрузки и сохранения с 6-разрядным непосредственным операндом без знака. Базовым регистром является фиксированный регистр, не указанный в коде команды, потому что он неявно определяется кодом операции загрузки/сохранения.

Форматы 5 и 6 предназначены для команд переходов и вызовов подпрограмм. Первый включает 12-разрядное непосредственное значение без знака, которое прибавляется к значению PC для вычисления целевого адреса. Второй расширяет смещение до 22 разрядов, при этом размер команды AVR увеличивается до 32 битов.

Адресация

Большинство команд работают с операндами, расположение которых необходимо каким-то образом указать. Этот механизм, который мы обсудим в данном разделе, называется **адресацией**.

Режимы адресации

До сих пор мы не рассказывали о том, как интерпретируются биты адресного поля для нахождения операнда. Самое время разобраться в этой проблеме. Итак, поговорим о **режимах адресации**.

Непосредственная адресация

Самый простой способ указания операнда — хранить в адресной части сам операнд, а не адрес операнда или какую-либо другую информацию, описывающую, где находится операнд. Такой операнд называется **непосредственным**, поскольку он автоматически вызывается из памяти одновременно с командой; следовательно, сразу становится непосредственно доступным. Один из вариантов команды с непосредственным адресом для загрузки в регистр R1 константы 4 показан на рис. 5.12.

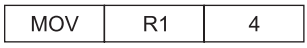


Рис. 5.12. Команда с непосредственным адресом для загрузки константы 4 в регистр 1

При непосредственной адресации не требуется дополнительного обращения к памяти для вызова операнда. Однако у такого способа адресации есть недостатки. Во-первых, таким способом можно работать только с константами. Во-вторых, число значений ограничено размером поля. Тем не менее эта технология используется во многих архитектурах для определения целочисленных констант.

Прямая адресация

Следующий способ определения операнда — просто дать его полный адрес. Такой режим называется **прямой адресацией**. Как и непосредственная адресация, прямая адресация имеет некоторые ограничения: команда всегда имет доступ только к одному и тому же адресу памяти. То есть значение может меняться,

а адрес — нет. Таким образом, прямая адресация может использоваться только для доступа к глобальным переменным, адреса которых известны во время компиляции. Многие программы содержат глобальные переменные, поэтому этот способ широко используется. Каким образом компьютер узнает, какие адреса непосредственные, а какие прямые, мы обсудим позже.

Регистровая адресация

Регистровая адресация по сути напоминает прямую, только в данном случае вместо ячейки памяти указывается регистр. Поскольку регистры очень важны (благодаря быстрому доступу и коротким адресам), этот режим адресации является самым распространенным на большинстве компьютеров. Многие компиляторы определяют, к каким переменным доступ будет осуществляться чаще всего (например, индексы циклов), и помещают эти переменные в регистры.

Такой режим называют **регистровой адресацией**. В архитектурах с перенаправлением для загрузки (например, в архитектуре ARM OMAP4420) практически все команды используют этот режим адресации. Он не применяется только в том случае, если операнд перемещается из памяти в регистр (команда `LDR`) или из регистра в память (команда `STR`). Но даже в этих командах один из операндов является регистром — туда отправляется слово из памяти и оттуда перемещается слово в память.

Косвенная регистровая адресация

При косвенной регистровой адресации искомый операнд берется из памяти или отправляется в память, но адрес не фиксируется жестко в команде, как при прямой адресации, а находится в регистре. Если адрес используется таким образом, он называется **указателем**. Преимущество косвенной адресации состоит в том, что можно обращаться к памяти, не имея в команде полного адреса. Кроме того, многократно выполняя данную команду, можно, меняя значение в регистре, использовать разные слова памяти.

Чтобы понять, почему может быть полезно использовать разные слова при каждом выполнении команды, представим себе цикл, который проходит по 1024-элементному одномерному массиву целых чисел для получения в регистре R1 суммы элементов. Вне этого цикла какой-то другой регистр, например R2, может указывать на первый элемент массива, а еще один регистр, например R3, — на первый адрес после массива. Массив содержит 1024 целых числа по 4 байта каждое. Если массив начинается с элемента *A*, то первый адрес после массива будет *A* + 4096. Типичная программа на ассемблере, выполняющая это вычисление для двухадресной машины, показана в листинге 5.1.

Листинг 5.1. Программа на ассемблере для вычисления суммы элементов массива

```
        MOV R1,#0           ; накопление суммы в R1, изначально 0
        MOV R2,#A           ; R2 = адрес массива A
        MOV R3,#A+4096      ; R3 = адрес первого слова после A
LOOP:   ADD R1,(R2)         ; получение операнда через регистр R2
        ADD R2,#4           ; увеличение R2 на одно слово(4 байта)
        CMP R2,R3           ; проверка завершения
        BLT LOOP            ; если R2 < R3, продолжать цикл
```

В этой маленькой программе мы использовали несколько режимов адресации. В первых трех командах выполняется регистровая адресация первого операнда (целевого) и непосредственная адресация второго (константа, обозначаемая символом #). Вторая команда помещает в R2 не содержимое элемента *A*, а *адрес* элемента *A*. Именно это и сообщает ассемблеру знак #. Сходным образом третья команда помещает в R3 первое слово после массива.

Интересно отметить, что само тело цикла не содержит каких-либо адресов памяти. В четвертой команде используются регистровая и косвенная адресация. В пятой команде применяются регистровая и непосредственная адресация, в шестой — оба раза регистровая. Команда `BLT` могла бы использовать адрес памяти, однако более привлекательным является определение адреса с помощью 8-разрядного смещения, связанного с самой командой `BLT`. Таким образом, вообще без обращения по адресам памяти мы получили короткий и быстрый цикл. Кстати, эта программа предназначена для Core i7, только мы переименовали команды и регистры и для простоты изменили запись, потому что синтаксис стандартного языка ассемблера Core i7 (MASM) является странным пережитком эпохи 8088.

Теоретически есть еще один способ выполнения этого фрагмента без косвенной регистровой адресации. Цикл мог бы содержать команду для прибавления *A* к регистру R1, например:

```
ADD R1, A
```

Тогда при каждом шаге команда должна увеличиваться на 4. Таким образом, после одного шага команда будет выглядеть следующим образом:

```
ADD R1, A + 4
```

И далее аналогично до завершения цикла.

Программа, которая сама изменяет себя подобным образом, называется **самомодифицирующейся программой**. Идея, предложенная еще Джоном фон Нейманом, применялась в старых компьютерах, которые не поддерживали режим косвенной регистровой адресации. В настоящее время самомодифицирующиеся программы считаются неудобными и трудными для понимания. Кроме того, их нельзя выполнять совместно несколькими процессорами. Они не могут правильно выполняться даже на машинах с разделенной кэш-памятью первого уровня, если в кэш-памяти команд нет специальной схемы для обратной записи (поскольку разработчики предполагали, что программы сами себя изменять не должны). Наконец, самомодифицирующиеся программы не будут работать на машинах с раздельными пространствами команд и данных. В целом эта идея осталась в прошлом (и это к лучшему).

Индексная адресация

Часто нужно уметь обращаться к словам памяти по известному смещению относительно регистра. Подобные примеры мы видели в машине IJVM, где локальные переменные определяются по смещению от регистра LV. Обращение к памяти по регистру и константе смещения называется **индексной адресацией**.

В машине IJVM при доступе к локальной переменной используется указатель ячейки памяти (LV) в регистре плюс небольшое смещение в самой команде, как показано на рис. 4.14, *a*. Есть и другой способ: указатель ячейки памяти в команде и небольшое смещение в регистре. Чтобы показать, как работает этот механизм,

рассмотрим следующий пример. Пусть у нас есть два одномерных массива A и B по 1024 слова в каждом. Нам нужно вычислить A_i И B_i для всех пар, а затем соединить все эти 1024 логических произведения операцией ИЛИ, чтобы узнать, есть ли в этом наборе хотя бы одна пара, не равная нулю. Один из вариантов — поместить адрес массива A в один регистр, а адрес массива B — в другой регистр, а затем последовательно перебирать элементы массивов, аналогично тому, как мы делали в предыдущей программе (см. листинг 5.1). Такая программа, конечно же, будет работать, но ее можно усовершенствовать, как показано в листинге 5.2.

Листинг 5.2. Программа на ассемблере, выполняющая операцию ИЛИ для 1024 элементов массива

```
        MOV R1,#0       ; собирает результаты выполнения ИЛИ в R1,
                        ; изначально 0
        MOV R2,#0       ; R2 = индекс, i от текущего
                        ; произведения A[i] И B[i]
        MOV R3,#4096    ; R3 = первое ненужное значение индекса

LOOP:   MOV R4,A(R2)    ; R4 = A[i]
        AND R4,B(R2)    ; R4 = A[i] И B[i]
        OR R1,R4
        ADD R2,#4       ; i = i + 4
        CMP R2,R3       ; нужно ли продолжать?
        BLT LOOP        ; если R2 < R3, нужно продолжать
```

Здесь нам требуется 4 регистра:

- R1 — содержит результаты суммирования логических произведений;
- R2 — индекс i, который используется для перебора элементов массива;
- R3 — константа 4096 (наименьшее неиспользуемое значение i);
- R4 — временный регистр для хранения каждого произведения.

После инициализации регистров мы входим в цикл из шести команд. Команда с меткой `LOOP` вызывает элемент A_i в регистр R4. При вычислении источника здесь используется индексная адресация. Регистр (R2) и константа (адрес элемента A) складываются, полученный результат служит для обращения к памяти. Сумма этих двух величин поступает в память, но не сохраняется ни в одном из доступных пользователю регистров. Следующая запись означает, что для определения приемника используется регистровая адресация, а для определения источника — индексная:

```
MOV R4,A(R2)
```

Здесь `R4` — это регистр, `A` — смещение, `R2` — регистр. Если `A` имеет значение, скажем, 124 300, то соответствующая машинная команда будет выглядеть так, как показано на рис. 5.13.

MOV	R4	R2	124300

Рис. 5.13. Возможное представление команды MOV R4, A(R2) в памяти

Во время первого прохождения цикла регистр R2 принимает значение 0 (так регистр инициализируется), поэтому нужное нам слово A_0 находится в ячейке с адресом 124 300. Это слово загружается в регистр R4. При следующем прохож-

дении цикла R2 принимает значение 4, поэтому нужное нам слово A_1 находится в ячейке с адресом 124 304 и т. д.

Как мы отмечали, здесь смещение — это указатель ячейки памяти, а значение регистра — это небольшое целое число, которое во время вычисления меняется. Такая форма требует, чтобы поле смещения в команде было достаточно большим для хранения адреса, поэтому такой способ не очень эффективен, однако он часто оказывается самым лучшим.

Относительная индексная адресация

В некоторых машинах применяется режим адресации, при котором адрес вычисляется путем суммирования значений двух регистров и смещения (смещение факультативно). Такой режим называется **относительной индексной адресацией**. Один из регистров — это база, другой — индекс. Относительная индексная адресация очень удобна при следующей ситуации. Вне цикла мы могли бы поместить адрес элемента A в регистр R5, а адрес элемента B — в регистр R6. Тогда можно было бы заменить две первые команды цикла `LOOP`:

```
LOOP:  MOV R4,(R2+R5)
       AND R4,(R2+R6)
```

Было бы идеально, если бы существовал режим адресации по сумме двух регистров без смещения. В то же время даже команда с 8-разрядным смещением была бы большим достижением, поскольку оба смещения можно сделать нулевыми. Однако если смещение всегда составляет 32 бита, тогда мы ничего не выиграем, используя такой режим адресации. На практике машины с относительной индексной адресацией обычно имеют форму с 8- или 16-разрядным смещением.

Стековая адресация

Мы уже отмечали, что очень желательно сделать машинные команды как можно короче. Предельный случай — команды без адресов. Как мы видели в главе 4, безадресные команды, например `IADD`, возможны при наличии стека. В этом подразделе мы рассмотрим стековую адресацию более подробно.

Обратная польская запись

В математике существует древняя традиция помещать оператор между операндами ($x + y$), а не после операндов ($x\ y\ +$). Форма с оператором между операндами называется **инфиксной записью**. Форма с оператором после операндов называется **постфиксной**, или **обратной польской записью** в честь польского логика Я. Лукасевича (1958), который изучал свойства этой записи.

Обратная польская запись имеет ряд преимуществ перед инфиксной записью при выражении алгебраических формул. Во-первых, любая формула может быть выражена без скобок. Во-вторых, она удобна для вычисления формул в машинах со стеками. В-третьих, инфиксные операторы имеют приоритеты, которые произвольны и нежелательны. Например, мы знаем, что $a \times b + c$, значит, $(a \times b) + c$, а не $a \times (b + c)$, поскольку произвольно было определено, что умножение имеет приоритет над сложением. Но имеет ли приоритет сдвиг влево над логической операцией И? Кто знает? Обратная польская запись позволяет устранить подобные неоднозначности.

Существуют несколько алгоритмов для превращения инфиксных формул в обратную польскую запись. Мы рассмотрим переработанный алгоритм, идея которого предложена Э. Дейкстра (E. W. Dijkstra). Предположим, что формула состоит из переменных, двухоперандных операторов +, –, *, /, а также левой и правой скобок. Чтобы отметить конец формулы, мы будем вставлять символ ⊥ после ее последнего символа и перед первым символом следующей формулы.

На рис. 5.14 нарисована железная дорога из Нью-Йорка в Калифорнию с ответвлением, ведущим в Техас. Каждый символ формулы представлен одним вагоном. Поезд движется на запад (налево). Перед стрелкой каждый вагон должен останавливаться и узнавать, должен ли он двигаться прямо в Калифорнию или ему нужно по пути заехать в Техас. Вагоны, содержащие переменные, всегда направляются в Калифорнию и никогда не едут в Техас. Вагоны, содержащие все прочие символы, должны перед прохождением стрелки узнавать о содержимом ближайшего вагона, отправившегося в Техас.

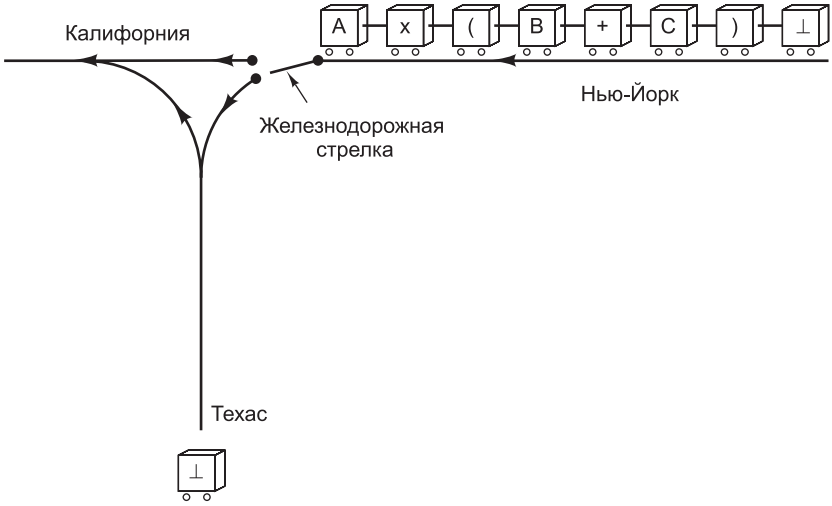


Рис. 5.14. Каждый вагон представляет собой один символ в формуле, которую нужно переделать из инфиксной формы в постфиксную

На рис. 5.15 показана зависимость ситуации от того, какой вагон отправился в Техас последним и какой вагон находится у стрелки. Первый вагон (помеченный символом ⊥) всегда отправляется в Техас. Числа соответствуют следующим ситуациям:

1. Вагон на стрелке направляется в Техас.
2. Последний вагон, направившийся в Техас, разворачивается и направляется в Калифорнию.
3. Вагон, находящийся на стрелке, и последний вагон, отправившийся в Техас, угоняются и исчезают (то есть, оба удаляются).
4. Остановка. Символы, находящиеся на калифорнийской ветке, представляют собой формулу в обратной польской записи, если читать слева направо.
5. Остановка. Произошла ошибка. Изначальная формула была некорректно сбалансирована.

Рис. 5.15. Алгоритм преобразования инфиксной записи в постфиксную

После каждого действия производится новое сравнение вагона, находящегося у стрелки (это может быть тот же вагон, что и в предыдущем сравнении, а может быть следующий вагон), и вагона, который на данный момент последним ушел на Техас. Этот процесс продолжается до тех пор, пока не будет достигнут шаг 4. Отметим, что линия на Техас используется как стек, где отправка вагона в Техас — это помещение элемента в стек, а разворот отправленного в Техас вагона в сторону Калифорнии — это выталкивание элемента из стека.

Порядок следования переменных в инфиксной и постфиксной записи одинаков. Однако порядок следования операторов не всегда один и тот же. В обратной польской записи операторы появляются в том порядке, в котором они будут выполняться. В табл. 5.5 даны примеры инфиксных формул и их эквивалентов в обратной польской записи.

Таблица 5.5. Некоторые примеры инфиксных выражений и их эквиваленты в обратной польской записи

Инфиксная запись	**Обратная польская запись**
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B)/(C − D)	A B + C D − /
A × B/C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Вычисление формул в обратной польской записи

Обратная польская запись идеально подходит для вычисления формул на компьютере со стеком. Формула состоит из *n* символов, каждый из которых является либо операндом, либо оператором. Алгоритм для вычисления формулы в обратную польской записи с использованием стека прост: строка с обратной польской записью сканируется слева направо. Если встречается операнд, его нужно поместить в стек. Если встречается оператор, нужно выполнить заданную им операцию.

Таблица 5.6 иллюстрирует вычисление машиной IJVM следующего выражения:

(8 + 2 × 5) / (1 + 3 × 2 − 4)

Соответствующая формула в обратной польской записи выглядит так:

8 2 5 × + 1 3 2 × + 4 – /

В таблице мы ввели команды умножения и деления `IMUL` и `IDIV`. Число на вершине стека — это правый операнд (а не левый). Это очень важно для операций деления и вычитания, поскольку порядок следования операндов в данном случае имеет значение (в отличие от операций сложения и умножения). Другими словами, команда `IDIV` действует следующим образом: сначала в стек помещается числитель, потом знаменатель, и тогда операция дает правильный результат. Отметим, что преобразовать обратную польскую запись в IJVM-код очень легко: нужно просто двигаться по формуле в обратной польской записи, записывая по одной команды для каждого символа. Если символ является константой или переменной, нужно вписывать команду помещения этой константы или переменной в стек, если символ является оператором, нужно вписывать команду для выполнения данной операции.

Таблица 5.6. Использование стека для вычисления формулы в обратной польской записи

Шаг	Оставшаяся цепочка	Команда	Стек
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Режимы адресации в командах перехода

До сих пор мы рассматривали только те команды, которые оперируют данными. Командам перехода (а также командам вызова процедур) также нужны особые режимы адресации для определения целевого адреса. Режимы адресации, о которых мы говорили в предыдущих подразделах, применимы и к большинству команд перехода. Один из возможных режимов — прямая адресация, когда целевой адрес просто полностью включается в команду.

Впрочем, существуют и другие полезные режимы адресации. Косвенная регистровая адресация позволяет программе вычислить целевой адрес, поместить

его в регистр, а затем перейти по полученному адресу. Такой способ дает максимальную гибкость, поскольку целевой адрес вычисляется во время выполнения программы. Но он также оставляет лазейку для бессчетного числа трудно обнаруживаемых ошибок.

Индексная адресация, при которой известно смещение от регистра, также вполне приемлема. Этот режим обладает теми же характеристиками, что и косвенная регистровая адресация.

Еще один режим — относительная адресация по счетчику команд. В данном случае для получения целевого адреса смещение (со знаком), находящееся в самой команде, прибавляется к счетчику команд. По сути, это индексная адресация, где в качестве регистра используется счетчик команд.

Ортогональность кодов операций и режимов адресации

С точки зрения программного обеспечения команды и режимы адресации должны иметь регулярную структуру с минимальным числом форматов команд. При такой структуре компилятору гораздо проще генерировать нужный код. Все коды операций должны поддерживать любые режимы адресации, если это имеет смысл. Более того, для всех регистровых режимов должны быть доступны все регистры, включая указатель кадра (FP), указатель стека (SP) и счетчик команд (PC).

Рассмотрим форматы 32-разрядных команд для трехадресной машины (рис. 5.16). Здесь поддерживаются до 256 кодов операций. В варианте 1 формата каждая команда имеет два входных регистра (источника) и один выходной регистр (приемник). Этот формат используется для всех арифметических и логических команд.

Рис. 5.16. Форматы команд для трехадресной машины

Неиспользованное 8-разрядное поле в конце команды может потребоваться для дальнейшей дифференциации команд. Например, можно иметь один код для всех операций с плавающей точкой, а различаться эти операции будут по дополнительному полю. Кроме того, если установлен бит 23, тогда задействуется вариант 2 формата, а второй операнд уже является не регистром, а 13-разрядной непосредственной константой со знаком. Команды `LOAD` и `STORE` тоже могут использовать этот формат для обращения к памяти при индексном режиме адресации.

Необходимо также иметь небольшое число дополнительных команд (например, команд условных переходов), но они легко подходят под вариант 3 формата. Например, можно приписать один код операции каждому (условному) переходу, вызову процедуры и т. д., тогда останется 24 бита для смещения по счетчику команд. Если предположить, что это смещение считается в словах, диапазон будет составлять ± 32 Мбайт. Несколько кодов операций можно зарезервировать для команд `LOAD` и `STORE`, которым нужны длинные смещения в варианте 3 формата. Отнести их к кодам общего назначения нельзя (например, команды `LOAD` и `STORE` будут выполняться только с R0), но и использоваться они будут редко.

Теперь рассмотрим структуру двухадресной машины, в которой в качестве любого операнда может использоваться слово из памяти (рис. 5.17). Такая машина умеет складывать слово памяти с регистром, регистр со словом памяти, два регистра и два слова памяти. В настоящее время обращение к памяти связано с относительно высокими затратами ресурсов, поэтому такая структура не очень распространена, но если с развитием технологий обращаться к памяти станет менее накладно, получится простое и эффективное решение. Машины PDP-11 и VAX, в которых использовались похожие форматы, были очень популярны и доминировали на рынке мини-компьютеров в течение двух десятилетий.

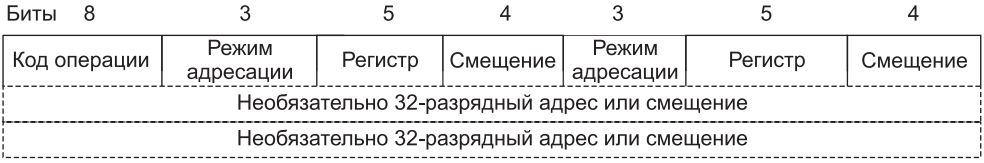


Рис. 5.17. Простые форматы команд для двухадресной машины

Здесь мы снова имеем 8-разрядный код операции, но теперь у нас есть по 12 бит для задания источника и приемника. Для каждого операнда 3 бита позволяют указать режим адресации, 5 бит — регистр и 4 бита — смещение. Имея 3 бита для задания режима адресации, мы можем поддерживать непосредственную, прямую, регистровую, косвенную регистровую индексную и стековую адресации и при этом еще остается место для двух дополнительных режимов, которые, возможно, появятся в будущем. Это простая система, которая легко компилируется и в то же время достаточно гибкая, особенно если счетчик команд, указатель стека и указатель локальных переменных находятся среди регистров общего назначения.

Единственная проблема то, что при прямой адресации требуется большее количество битов для адреса. В машинах PDP-11 и VAX к команде было добавлено дополнительное слово для указания адреса каждого прямо адресуемого операнда. Мы тоже могли бы использовать один из двух доступных режимов адресации для индексной адресации с 32-разрядным смещением, которое следует за командой. Тогда в худшем случае при сложении двух слов памяти, когда обращение к обоим операндам производится в режиме прямой адресации или с использованием длинной индексной формы, команда была бы размером 96 бит и занимала бы 3 цикла шины (один для команды и два для данных). В то же время для прибавления произвольного слова памяти к другому произвольному слову памяти большинству RISC-систем потребовалось бы по крайней мере 96 бит, а может и больше, на что нужно, по крайней мере, 4 цикла шины (в зависимости от способа адресации операндов).

У форматов, изображенных на рис. 5.17, существует много альтернатив. В данной системе с помощью одной 32-разрядной команды, при условии что переменные `i` и `j` находятся среди первых 16 локальных переменных, можно выполнить следующую операцию:

```
i = j;
```

Для переменных, расположенных после первых 16, нам потребуется перейти к 32-разрядным смещениям. Можно также придумать формат с одним 8-разрядным смещением вместо двух 4-разрядных при условии, что это смещение может использоваться либо источником, либо приемником, но не обоими. Существует множество решений с разными достоинствами и недостатками, и для получения хорошего результата проектировщику приходится учитывать многие факторы.

Режимы адресации процессора Core i7

Режимы адресации процессора Core i7 чрезвычайно нерегулярны и зависят от формата конкретной команды — 16- или 32-разрядная. Мы не будем рассматривать 16-разрядные команды, вполне достаточно 32-разрядных. Поддерживаемые режимы адресации включают непосредственную, прямую, регистровую, косвенную регистровую, индексную и специальную адресацию для обращения к элементам массива. Проблема заключается в том, что не все режимы применимы ко всем командам и не все регистры могут использоваться во всех режимах адресации. Это значительно усложняет задачу разработчика компилятора.

Как показано на рис. 5.10, для управления режимами адресации имеется соответствующий байт. Один из операндов определяется по комбинации полей MOD и R/M. Второй операнд всегда является регистром и определяется по значению поля REG. В табл. 5.7 приведен список 32 комбинаций значений 2-разрядного поля MOD и 3-разрядного поля R/M. Например, если оба поля равны 0, операнд считывается из ячейки памяти с адресом, который содержится в регистре EAX.

Таблица 5.7. 32-разрядные режимы адресации процессора Core i7 (M[x] — это слово в памяти с адресом x)

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + смещение 8]	M[EAX + смещение 32]	EAX или AL
001	M[ECX]	M[ECX + смещение 8]	M[ECX + смещение 32]	ECX или CL
010	M[EDX]	M[EDX + смещение 8]	M[EDX + смещение 32]	EDX или DL
011	M[EBX]	M[EBX + смещение 8]	M[EBX + смещение 32]	EBX или BL
100	SIB	SIB со смещением 8	SIB со смещением 32	ESP или AH
101	Прямая адресация	M[EBP + смещение 8]	M[EBP + смещение 32]	EBP или CH
110	M[ESI]	M[ESI + смещение 8]	M[ESI + смещение 32]	ESI или DH
111	M[EDI]	M[EDI + смещение 8]	M[EDI + смещение 32]	EDI или BH

Столбцы 01 и 10 включают режимы адресации, при которых значение регистра прибавляется к 8- или 32-разрядному смещению, следующему за ко-

мандой. Если выбрано 8-разрядное смещение, оно перед сложением получает 32-разрядное знаковое расширение. Например, команда `ADD` с полем R/M = 011, полем MOD = 01 и смещением, равным 6, вычисляет сумму регистра EBX и 6 и в качестве одного из операндов считывает слово из полученного адреса памяти. Значение регистра EBX не изменяется.

При MOD = 11 предоставляется возможность выбора из двух регистров. Для команд со словами берется первый вариант, для команд с байтами — второй. Отметим, что здесь не все регулярно. Например, нельзя выполнить косвенную адресацию через EBP или прибавить смещение к ESP.

Иногда вслед за байтом MODE следует дополнительный байт SIB (см. рис. 5.9). Байт SIB определяет масштабный коэффициент и два регистра. При наличии байта SIB адрес операнда вычисляется путем умножения индексного регистра на 1, 2, 4 или 8 (в зависимости от значения поля SCALE), прибавлением его к базовому регистру и, наконец, возможным прибавлением 8- или 32-разрядного смещения, в зависимости от значения поля MOD. Практически все регистры могут использоваться и в качестве индекса, и в качестве базы.

Режимы, получаемые посредством байта SIB, могут пригодиться для обращения к элементам массива. Рассмотрим следующую Java-команду:

```
for (i=0; i<n; i++) a[i]=0;
```

Здесь `a` — массив 4-байтных целых чисел, относящийся к текущей процедуре. Обычно регистр EBP используется для указания на базу стекового кадра, который содержит локальные переменные и массивы, как показано на рис. 5.18. Компилятор должен хранить значение `i` в регистре EAX. Для доступа к элементу `a[i]` он будет использовать режим с байтом SIB, в котором адрес операнда равен сумме значений 4 × EAX, EBP и 8. Эта операция может сохраняться в `a[i]` за одну команду.

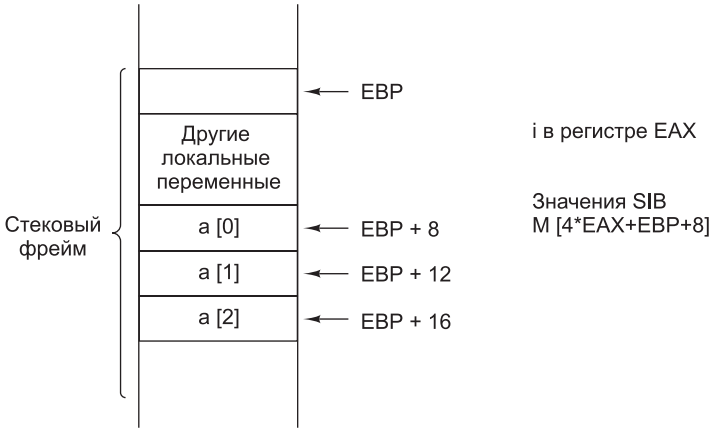


Рис. 5.18. Обращение к элементу массива a[i]

А стоит ли применять такой режим адресации? Трудно сказать. Безусловно, эта команда при надлежащем использовании позволяет сэкономить несколько циклов. С другой стороны, она занимает определенную область памяти микросхемы, которую можно было бы использовать для других целей — например, для увеличения размера кэш-память первого уровня или уменьшения размера микросхемы с возможным повышением тактовой частоты.

Такие компромиссные решения проектировщику приходится принимать постоянно. Обычно перед тем как воплотить какую-либо идею в кремнии, выполняются многочисленные моделирующие прогоны, но для этого нужно иметь представление о том, как будет использоваться машина. Конечно, разработчики процессора 8088 не включали веб-браузер в свой набор тестов, и решения, принятые 20 лет назад, могут оказаться абсолютно неудачными с точки зрения современных приложений. В то же время, включив в машину какую-нибудь функцию, из-за требования обратной совместимости избавиться от нее будет уже невозможно.

Режимы адресации процессора OMAP4430

В архитектуре команд процессора OMAP4430 все команды поддерживают режим непосредственной или регистровой адресации, за исключением тех команд, которые обращаются к памяти. При регистровой адресации 5 бит просто сообщают, какой регистр нужно использовать. При непосредственной адресации в качестве данных используется 12-разрядная константа без знака. Для арифметических, логических и других подобного рода команд никаких других режимов адресации не используется.

К памяти обращаются команды двух типов: загрузки (`LOAD`) и сохранения (`STORE`). Они поддерживают три режима адресации. В первом режиме вычисляется сумма двух регистров, а затем полученное значение используется для косвенной адресации. Во втором режиме адрес вычисляется как сумма базового регистра с 13-разрядным смещением со знаком. В третьем режиме адрес вычисляется как сумма регистра PC с 13-разрядным смещением со знаком. В частности, этот режим хорошо подходит для загрузки программных констант, хранящихся в коде программы.

Режимы адресации процессора ATmega168

Схема адресации процессора ATmega168 достаточно стандартна. Предусмотрено четыре основных режима адресации. Первый — регистровый режим, при котором операнды хранятся в регистре. Регистры могут использоваться как для источника, так и для приемника. Во втором — непосредственном — режиме 8-разрядное непосредственное значение без знака кодируется в команде.

Остальные режимы используются только командами загрузки и сохранения. Третий режим — прямая адресация, когда операнд находится в ячейке памяти, адрес которой содержится в команде. Для 16-разрядных команд прямая адресация ограничивается 7 битами (следовательно, загружаться могут только адреса от 0 до 127). В архитектуре AVR также определяется 32-разрядная команда, которая содержит 16-разрядный адрес, поддерживающий до 64 Кбайт памяти.

Четвертый режим — косвенная регистровая адресация, при которой в регистре хранится указатель на операнд. Поскольку разрядность обычных регистров составляет 8 бит, команды загрузки и сохранения используют пары регистров для представления 16-разрядных адресов. Пара регистров может адресовать до 64 Кбайт памяти. Архитектура поддерживает использование трех пар регистров: X, Y и Z, состоящих из регистров R26/R27, R28/R29 и R30/R31 соответственно. Например, чтобы загрузить адрес в регистр X, программа должна загрузить 8-разрядное значение в регистры R26 и R27, что потребует двух команд.

Сравнение режимов адресации

Рассмотренные режимы адресации машин Core i7, OMAP4430 и ATmega168 иллюстрирует табл. 5.8. Как мы уже отмечали, в командах может использоваться не каждый режим.

Таблица 5.8. Сравнение режимов адресации

Режим адресации	Core i7	OMAP4430	ATmega168 AVR
Непосредственная	Да	Да	Да
Прямая	Да		Да
Регистровая	Да	Да	Да
Косвенная регистровая	Да	Да	Да
Индексная	Да	Да	
Относительная индексная		Да	

На практике для эффективной архитектуры команд вовсе не нужно множество разнообразных режимов адресации. Поскольку практически весь код, написанный на этом уровне, генерируется компиляторами (за исключением ATmega168), режимов адресации должно быть мало и они должны быть простыми и понятными. Машине приходится занимать одну из крайних позиций: поддерживать либо все доступные варианты, либо только один вариант. В промежуточных случаях компилятор сталкивается с необходимостью выбора, сделать который он не в силах ввиду недостатка информации или сложности алгоритма.

Поэтому в наиболее простых архитектурах используют очень немного режимов адресации, причем на каждый из этих режимов накладываются жесткие ограничения. Обычно практически для любых применений достаточно непосредственной, прямой, регистровой и индексной адресации. Каждый регистр (включая указатель локальных переменных, указатель стека и счетчик команд) должен быть доступен всегда, когда это требуется. В более сложных режимах адресации можно сократить число команд, но при этом придется ввести жесткие последовательности операций, которые трудно будет выполнять параллельно с другими операциями.

Мы рассмотрели возможные варианты организации кодов операций и адресов, а также различные режимы адресации. Когда вы сталкиваетесь с новым компьютером, нужно изучить все его команды и режимы адресации, причем не только для того, чтобы знать об их существовании, но и чтобы понять, почему разработчиками был сделан именно такой выбор и каковы были бы последствия при другом выборе.

Типы команд

Команды можно условно разделить на несколько групп, которые, хотя и могут различаться в деталях, воспроизводятся практически в каждой машине. Кроме того, в каждом компьютере всегда имеются несколько необычных команд, добавленных либо в целях совместимости с предыдущими моделями, либо потому,

что у разработчика возникла блестящая идея, либо по требованию правительства, заплатившего производителю, чтобы тот включил в набор команд новую команду. В этом разделе мы попытаемся описать все наиболее распространенные категории, однако отметим, что мы не претендуем на исчерпывающее изложение.

Команды перемещения данных

Копирование данных из одного места в другое — одна из самых распространенных операций. Под копированием мы понимаем создание нового объекта с точно таким же набором битов, как у исходного. Такое понимание слова «перемещение» несколько отличается от его обычного значения. Если мы говорим, что какой-то человек переместился из Нью-Йорка в Калифорнию, это не значит, что в Калифорнии была создана идентичная копия этого человека, а оригинал остался в Нью-Йорке. Когда мы говорим, что содержимое ячейки памяти 2000 переместилось в какой-либо регистр, мы всегда подразумеваем, что там была создана идентичная копия, а оригинал все еще находится в ячейке 2000. Команды перемещения данных лучше было бы назвать командами дублирования данных, но название уже устоялось.

Есть две причины, по которым данные могут копироваться из одного места в другое. Одна из них фундаментальна: присваивание переменным значений. Следующая операция присваивания выполняется путем копирования значения, которое находится в ячейке памяти с адресом *B*, в ячейку *A*, поскольку программист приказал это сделать:

```
A = B
```

Вторая причина копирования данных — скорость обращения и эффективность использования. Как мы уже видели, многие команды могут обращаться к переменным только в том случае, если они находятся в регистре. Поскольку существует два возможных источника данных (память и регистр) и два возможных приемника данных (память и регистр), существует также 4 различных способа копирования. В одних компьютерах для этих четырех случаев поддерживаются 4 команды, в других — единственная команда. Некоторые компьютеры используют команду `LOAD` для загрузки данных из памяти в регистр, команду `STORE` для сохранения в памяти данных из регистра, команду `MOVE` для перемещения данных из одного регистра в другой, но вообще не имеют команд для копирования из одной части памяти в другую.

Команды перемещения данных должны как-то указывать, сколько данных нужно переместить. Существуют команды для перемещения разных объемов данных — от одного бита до всей памяти. В машинах с фиксированной длиной слова обычно перемещается ровно одно слово. Любые перемещения других объемов данных (больше или меньше слова) должны выполняться программно с использованием операций сдвига и слияния. Некоторые архитектуры команд дают возможность копировать фрагменты данных размером меньше слова (они обычно измеряются в байтах), а также сразу нескольких слов. Копирование нескольких слов рискованно, особенно если максимальное количество слов достаточно большое, поскольку такая операция может занять много времени, и существует вероятность ее прерывания в середине. Некоторые машины с переменной длиной слов содержат команды, которые определяют только адреса источника

и приемника, но не объем данных — в этом случае перемещение продолжается до тех пор, пока не встретится специальное поле, содержащее признак окончания данных.

Бинарные операции

Бинарные операции вычисляют результат на основе значений двух операндов. Все архитектуры команд содержат команды для сложения и вычитания целых чисел. Кроме того, практически в архитектурах имеются команды умножения и деления целых чисел. Вероятно, нет необходимости объяснять, почему компьютеры оснащены арифметическими командами.

Следующую группу бинарных операций составляют булевы команды. Хотя существует 16 булевых функций от двух переменных, команды для всех 16 поддерживаются в очень немногих машинах. Обычно поддерживаются только операции И, ИЛИ и НЕ; иногда кроме них еще ИСКЛЮЧАЮЩЕЕ ИЛИ, НЕ-ИЛИ и НЕ-И.

Важным применением операции И является выделение битов из слов. Рассмотрим машину со словами длиной в 32 бита, в которой на одно слово приходится четыре 8-разрядных символа. Предположим, что нужно отделить второй символ от остальных трех, чтобы его напечатать. Это значит, что нужно создать слово, в котором этот символ займет правые 8 бит, а левые 24 бит должны стать нулевыми (так называемое **выравнивание вправо**).

Чтобы извлечь нужный нам символ, слово, содержащее этот символ, соединяется операцией И с константой, которая называется **маской**. В результате этой операции все ненужные биты меняются на нули.

A:

```
10110111 10111100 11011011 10001011
```

B (маска):

```
00000000 11111111 00000000 00000000
```

A И *B*:

```
00000000 10111100 00000000 00000000
```

Затем результат сдвигается на 16 бит вправо, чтобы нужный символ оказался в правой части слова.

Важным применением команды ИЛИ является помещение битов в слово. Эта операция обратна операции извлечения. Чтобы изменить правые 8 бит 32-разрядного слова, не повредив при этом остальные 24 бита, сначала нежелательные 8 бит надо заменить нулями, а затем новый символ соединить операцией ИЛИ с полученным результатом.

A:

```
10110111 10111100 11011011 10001011
```

B (маска):

```
11111111 11111111 11111111 00000000
```

A И *B*:

```
10110111 10111100 11011011 00000000
```

C:

```
00000000 00000000 00000000 01010111
```

(*A* И *B*) ИЛИ *C*:

```
10110111 10111100 11011011 01010111
```

Операция И удаляет единицы, поэтому в полученном результате никогда не бывает больше единиц, чем в любом из двух операндов. Операция ИЛИ вставляет единицы, поэтому в полученном результате всегда по крайней мере столько же единиц, сколько в операнде с большим количеством единиц. Операция ИСКЛЮЧАЮЩЕЕ ИЛИ, в отличие от них, симметрична в отношении единиц и нулей. Такая симметрия иногда может быть полезной, например, при генерировании случайных чисел.

Большинство современных компьютеров поддерживает команды с плавающей точкой, которые в основном соответствуют арифметическим операциям с целыми числами. Большая часть машин поддерживает по крайней мере два варианта таких чисел: более короткие для скорости и более длинные на тот случай, если требуется высокая точность вычислений. Существуют множество возможных форматов для чисел с плавающей точкой, но сейчас практически везде применяется единый стандарт IEEE 754. Числа с плавающей точкой и этот стандарт обсуждаются в приложении Б.

Унарные операции

В унарных операциях результат получается обработкой единственного операнда. Поскольку в данном случае нужно задавать на один адрес меньше, чем в бинарных операциях, команды иногда бывают короче, если только не требуется задавать другую информацию.

Очень полезны команды сдвига и циклического сдвига. Они часто даются в нескольких вариантах. Сдвиги — это операции, при которых биты сдвигаются влево или вправо, при этом биты, которые сдвигаются за пределы слова, утрачиваются. Циклические сдвиги — это сдвиги, при которых биты, вытесненные с одного конца, появляются на другом конце слова. Разницу между обычным сдвигом и циклическим сдвигом иллюстрирует следующий пример:

A:

```
00000000 00000000 00000000 01110011
```

Сдвиг *A* вправо на 2 бита:

```
00000000 00000000 00000000 00011100
```

Циклический сдвиг *A* вправо на 2 бита:

```
11000000 00000000 00000000 00011100
```

Обычные и циклические сдвиги влево и вправо играют очень важную роль. Если n-разрядное слово циклически сдвигается влево на k бит, результат будет такой же, как при циклическом сдвиге вправо на $n - k$ бит.

Сдвиги вправо часто выполняются с расширением по знаку. Это значит, что позиции, освободившиеся на левом конце слова, заполняются знаковым битом (0 или 1) исходного слова, как будто знаковый бит смещается вправо. Кроме

того, это значит, что отрицательное число остается отрицательным. Вот как выглядят сдвиги на 2 бита вправо:

A:

```
1111111 11111111 11111111 11110000
```

A сдвигается без знакового расширения:

```
0011111 11111111 11111111 11111100
```

A сдвигается со знаковым расширением:

```
1111111 11111111 11111111 11111100
```

Операция сдвига используется при умножении и делении на 2. Если положительное целое число сдвигается влево на k бит, результатом будет исходное число, умноженное на 2^k. Если положительное целое число сдвигается вправо на k бит, то результатом становится исходное число, деленное на 2^k.

Сдвиги могут использоваться для ускорения некоторых арифметических операций. Рассмотрим выражение $18 \times n$, где n — положительное целое число. $18 \times n = 16 \times n + 2 \times n$. Значение $16 \times n$ можно получить путем сдвига копии n на 4 бита влево. Значение $2 \times n$ можно получить, сдвинув n на 1 бит влево. Сумма этих двух чисел равна $18 \times n$. Таким образом, целиком произведение можно вычислить путем одного перемещения, двух сдвигов и одного сложения, что обычно выполняется гораздо быстрее, чем операция умножения. Конечно, компилятор может применить этот прием, только если один из множителей является константой.

Сдвиг отрицательных чисел даже со знаковым расширением дает совершенно другие результаты. Рассмотрим, например, число –1 в обратном двоичном коде. При сдвиге влево на 1 бит получается число –3. При сдвиге влево еще на 1 бит получается число –7:

Число –1 в обратном двоичном коде:

```
11111111 11111111 11111111 11111110
```

Число –1 сдвигается влево на 1 бит (–3):

```
11111111 11111111 11111111 11111100
```

Число –1 сдвигается влево на 2 бита (–7):

```
11111111 11111111 11111111 11111000
```

Сдвиг влево отрицательных чисел в обратном двоичном коде не ведет к умножению числа на 2. В то же время сдвиг вправо корректно обеспечивает деление.

А теперь рассмотрим число –1 в дополнительном двоичном коде. При сдвиге вправо на 6 бит с расширением по знаку получается число –1, что неверно, поскольку целая часть от –1/64 равна 0:

Число –1 в дополнительном двоичном коде:

```
11111111 11111111 11111111 11111111
```

Число –1, сдвинутое влево на 6 бит, равно –1:

```
11111111 11111111 11111111 11111111
```

Как правило, сдвиг вправо вызывает ошибки, поскольку он производит округление до большего отрицательного целого числа, что недопустимо при выполнении целочисленных арифметических операций с отрицательными числами. В то же время сдвиг влево аналогичен умножению на 2.

Операции циклического сдвига нужны для манипулирования последовательностями битов в словах. Чтобы проверить все биты в слове, каждый бит путем циклического сдвига слова поочередно помещается в знаковый разряд, где его можно легко проверить; когда все биты будут проверены, восстанавливается изначальное значение слова. Операции циклического сдвига гораздо удобнее операций обычного сдвига, поскольку при этом не теряется информация: любую операцию циклического сдвига можно отменить операцией циклического сдвига в противоположную сторону.

Некоторые бинарные операции так часто используются с конкретными значениями операндов, что для их быстрого выполнения в архитектуры команд часто включаются унарные операции. Например, в начале вычислений чрезвычайно часто выполняется запись нуля в память или в регистр. Перемещение нуля — это особый случай команды перемещения данных. Поэтому для повышения производительности в архитектуру вводится операция `CLR` с единственным операндом — адресом той ячейки, которую нужно очистить (то есть установить на 0).

Прибавление единицы к слову тоже часто требуется при различных подсчетах. Унарная форма команды `ADD` — это операция инкремента `INC`. Другой пример — операция `NEG`. Отрицание X — это на самом деле бинарная операция вычитания $0 - X$, но поскольку операция отрицания применяется очень часто, в архитектуру команд вводится команда `NEG`. Важно понимать разницу между арифметической операцией `NEG` и логической операцией `NOT`. При выполнении операции `NEG` происходит **аддитивная инверсия** числа (такое число, сумма которого с исходным числом дает 0). При выполнении операции `NOT` все биты в слове просто инвертируются. Эти операции очень похожи, а для системы, в которой отрицательные величины представлены в обратном двоичном коде, они идентичны. (В арифметике дополнительных кодов для выполнения команды `NEG` сначала инвертируются все биты, а затем к полученному результату прибавляется 1.)

Унарные и бинарные операции часто объединяются в группы по функциям, которые они выполняют, а вовсе не по числу операндов. В первую группу входят арифметические операции, в том числе операция отрицания. Во вторую группу входят логические операции и операции сдвига, поскольку эти две категории очень часто используются вместе для извлечения данных.

Сравнения и условные переходы

Практически все программы должны проверять свои данные и на основе полученных результатов изменять последовательность выполняемых команд. Для примера возьмем функцию извлечения квадратного корня из числа x. Если это число отрицательно, функция должна сообщать об ошибке, если положительно — выполнять вычисления. То есть соответствующая программа должна проверять знак числа x, а затем совершать переход в зависимости от полученного результата.

Этот алгоритм можно реализовать с помощью специальных команд условного перехода, проверяющих разного рода условия и при их выполнении совершаю-

щих переходы к определенным адресам памяти. Иногда один из битов в команде указывает, нужно ли осуществлять переход в случае выполнения или в случае не выполнения условия соответственно. Часто целевой адрес является не абсолютным, а относительным (связанным с текущей командой).

Самое распространенное условие — проверка на равенство или на неравенство определенного бита нулю. Если команда проверяет знаковый бит числа и совершает переход к метке (`LABEL`), когда этот бит равен 1 (проверяемое число отрицательно), то выполняются те команды, которые начинаются с метки `LABEL`, а если этот бит равен 0 (проверяемое число положительно), то выполняются команды, следующие за командой условного перехода.

Во многих машинах имеются биты, обозначающие конкретные условия; например, это может быть бит переполнения, который принимает значение 1 всякий раз, когда арифметическая операция выдает неправильный результат. По этому биту проверяется правильность выполнения предыдущей арифметической операции, и в случае ошибки запускается программа обработки ошибок.

В некоторых процессорах есть специальный бит переноса, который принимает значение 1, если происходит перенос из самого левого бита (например, при сложении двух отрицательных чисел). Бит переноса нельзя путать с битом переполнения. Проверка бита переноса необходима для вычислений с повышенной точностью (то есть когда целое число представлено двумя или более словами).

Проверка на ноль очень важна при выполнении циклов и в некоторых других случаях. Если бы все команды условного перехода проверяли только 1 бит, то тогда для проверки определенного слова на равенство 0 нужно было бы поочередно проверять каждый бит, чтобы убедиться, что ни один из них не равен 1. Чтобы избежать подобной ситуации, многие машины поддерживают команду, которая проверяет слово целиком и выполняет переход, если оно равно 0. Конечно же, в этом решении все равно проверяется каждый бит, просто ответственность за проверку перекладывается на микроархитектуру. На практике в число устройств обычно включается регистр, все биты которого соединяются операцией ИЛИ, чтобы получить в результате бит, показывающий, имеются ли в регистре единичные биты. Бит Z на рис. 4.1 обычно вычисляется следующим образом: сначала все выходные биты АЛУ соединяются операцией ИЛИ, а затем полученный результат инвертируется.

Операция сравнения слов или символов очень важна, например, при сортировке. Чтобы выполнить сравнение, требуются три адреса: два нужны для данных, а по третьему адресу будет совершаться переход в случае выполнения условия. В тех компьютерах, где форматы команд позволяют иметь три адреса в команде, проблем не возникает. Но если такие форматы не предусмотрены, нужно что-то делать, чтобы обойти проблему.

Одно из возможных решений — ввести команду, которая выполняет сравнение и записывает результат в виде одного или нескольких битов условий. Следующая команда может проверять биты условия и совершать переход, если два сравниваемых значения были равны, или если они были неравны, или если первое из них было больше второго и т. д. Такой подход применяется в Core i7, OMAP4430 и ATmega168 AVR.

В операции сравнения двух чисел есть некоторые тонкости. Сравнение — это не такая простая операция, как вычитание. Если очень большое положительно

число сравнивается с очень большим отрицательным числом, операция вычитания приведет к переполнению, поскольку результат вычитания будет невозможно представить. Тем не менее команда сравнения и в этом случае должна определить, удовлетворяется ли условие, и возвратить правильный ответ. При сравнении переполнения возникать не должно.

Кроме того, при сравнении чисел нужно решить, считать ли числа числами со знаком или числами без знака. 3-разрядные бинарные числа можно упорядочить двумя способами. От самого маленького к самому большому:

- Без знака: 000, 001, 010, 011, 100, 101, 110, 111.
- Со знаком: 100, 101, 110, 111, 000, 001, 010, 011.

В первой строке по возрастанию перечислены положительные числа от 0 до 7, во второй — целые числа со знаком от –4 до +3 в дополнительном двоичном коде (тоже по возрастанию). Ответ на вопрос: «Какое число больше: 011 или 100?» — зависит от того, считаются ли числа числами со знаком. В большинстве архитектур есть команды для обработки обоих вариантов упорядочения.

Команды вызова процедур

Процедурой называют группу команд, которая решает определенную задачу и которую можно вызывать из разных мест программы. Вместо термина процедура часто используется термин **подпрограмма**, особенно когда речь идет о программах на языке ассемблера. В языке C процедуры называются **функциями**, даже если они не являются функциями в математическом смысле. В Java используется термин «**метод**». Когда процедура заканчивает решение задачи, она должна вернуться к оператору, расположенному в программе следом за оператором вызова процедуры. Следовательно, адрес возврата должен как-то передаваться процедуре или сохраняться где-либо таким образом, чтобы можно было определить, куда возвращаться после решения задачи.

Адрес возврата может помещаться в одном из трех мест: в памяти, в регистре или в стеке. Самое худшее решение — поместить этот адрес в фиксированную ячейку памяти. Тогда, если процедура вызовет другую процедуру, второй вызов приведет к потере первого адреса возврата.

Более удачное решение — сохранить адрес возврата в первом слове процедуры, а первой выполняемой командой сделать второе слово процедуры. После завершения процедуры будет происходить переход к первому слову, а если аппаратно в первом слове наряду с адресом возврата предоставить код операции, произойдет непосредственный переход к этой операции. Процедура может вызывать другие процедуры, поскольку в каждой процедуре имеется пространство для одного адреса возврата. Но если процедура вызывает сама себя, эта схема не сработает, поскольку первый адрес возврата будет уничтожен вторым вызовом. (Способность процедуры вызывать саму себя, называемая **рекурсией**, очень важна и теоретически, и практически.) Более того, если процедура *A* вызывает процедуру *B*, процедура *B* вызывает процедуру *C*, а процедура *C* вызывает процедуру *A* (непосредственная, или цепочечная, рекурсия), эта схема сохранения адреса возврата также не сработает. Схема сохранения адреса возврата в первом слове процедуры использовалась в CDC 6600 — самом быстром компьютере

в мире в конце 1960-х годов. Основным языком 6600 был FORTRAN, в котором рекурсия запрещена, так что тогда решение работало. Тем не менее и тогда, и сейчас это решение нельзя признать удовлетворительным.

Более удачное решение — сохранение адреса возврата в регистре. В этом случае ответственность за сохранение его в безопасном месте возлагается на саму процедуру. Если процедура рекурсивна, ей придется помещать адрес возврата в новое место при каждом вызове.

Самое лучшее решение — поместить адрес возврата в стек. Тогда при завершении процедуры она должны выталкивать адрес возврата из стека. При такой форме вызова процедур рекурсия не порождает никаких проблем; адрес возврата будет автоматически сохраняться таким образом, чтобы не уничтожить предыдущий адрес возврата. Мы рассматривали такой способ сохранения адреса возврата в машине IJVM (см. рис. 4.9).

Управление циклами

Часто возникает необходимость выполнять некоторую группу команд фиксированное количество раз, поэтому некоторые машины для управления этим процессом содержат специальные команды. Во всех схемах подобного рода имеется счетчик, который увеличивается или уменьшается на какую-либо константу каждый раз при выполнении цикла. Кроме того, этот счетчик каждый раз проверяется. В случае удовлетворения проверяемого условия цикл завершается.

Счетчик запускается вне цикла, и затем сразу начинается выполнение цикла. Последняя команда цикла обновляет счетчик, и если условие завершения цикла еще не удовлетворено, происходит возврат к первой команде цикла. Если условие удовлетворено, цикл завершается и начинается выполнение команды, расположенной сразу после цикла. Цикл такого типа с проверкой в конце цикла представлен в листинге 5.3. (Мы не могли здесь использовать язык Java, поскольку в нем нет оператора `goto`.)

Листинг 5.3. Цикл с проверкой в конце

```
        i = 1;
L1:     первый оператор;
        .
        .
        .
        последний оператор;
        i = i + 1;
        if (i<n) goto L1;
```

Цикл такого типа всегда будет выполнен хотя бы один раз, даже если `n` ≤ 0. Представим программу, которая поддерживает базу данных о персонале компании. В определенном месте программа начинает считывать информацию о конкретном работнике. Она считывает число `n` — количество детей у работника, и выполняет цикл `n` раз, по одному разу на каждого ребенка. В цикле она считывает его имя, пол и дату рождения, так что компания сможет послать ему (или ей) подарок точно в срок. Однако если у работника нет детей и значение `n` равно 0, цикл все равно будет выполнен один раз, что даст ошибочные результаты.

В листинге 5.4 представлен другой способ проверки, который дает правильные результаты даже при n ≤ 0. Отметим, что если и увеличение счетчика, и проверка условия выполняются в одной команде, разработчикам придется выбирать один из двух методов.

Листинг 5.4. Цикл с проверкой в начале

```
        i = 1;
L1:     if(i>n) goto L2;
        первый оператор;
        .
        .
        .
        последний оператор;
        i = i + 1;
        goto L1;
L2:
```

Рассмотрим код, который будет сгенерирован компилятором при обработке следующей строки:

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

Если у компилятора нет никакой информации о числе n, ему, чтобы корректно обработать случай n ≤ 0, придется действовать так, как показано в листинге 5.4. Однако если компилятор определит, что n > 0 (например, проверив, какое значение присвоено n), он сможет использовать более эффективный код, представленный в листинге 5.3. Когда-то в стандарте языка FORTRAN требовалось, чтобы все циклы выполнялись хотя бы один раз. Это позволяло всегда порождать более эффективный код (как в листинге 5.3). В 1977 году этот дефект был исправлен, поскольку даже приверженцы языка FORTRAN начали осознавать, что не слишком хорошо иметь оператор цикла с такой странной семантикой, хотя она и позволяет экономить одну команду перехода на каждом цикле. В C и Java всегда использовался нормальный подход.

Команды ввода-вывода

Ни одна другая группа команд не различается в разных машинах так сильно, как команды ввода-вывода. В современных персональных компьютерах используются три разные схемы ввода-вывода:

- программируемый ввод-вывод с активным ожиданием;
- ввод-вывод с управлением по прерываниям;
- ввод-вывод с прямым доступом к памяти.

Мы рассмотрим каждую из этих схем по очереди.

Самым простым методом ввода-вывода является **программируемый ввод-вывод**. Эта схема часто используется в дешевых микропроцессорах, например во встроенных системах или в таких системах, которые должны быстро реагировать на внешние изменения (системы реального времени). Подобные процессоры обычно имеют одну команду ввода и одну команду вывода. Каждая из этих команд выбирает одно из устройств ввода-вывода. Между фиксированным регистром процессора и выбранным устройством ввода-вывода передается по

одному символу. Процессор должен выполнять определенную последовательность команд при каждом считывании и записи символа.

В качестве примера рассмотрим терминал с четырьмя 1-байтными регистрами, как показано на рис. 5.19. Два регистра используются для ввода: регистр состояния устройства и регистр данных. Два регистра используются для вывода: тоже регистр состояния устройства и регистр данных. Каждый из них имеет уникальный адрес. Если имеет место ввод-вывод с отображением на память, все 4 регистра являются частью адресного пространства и могут считываться и записываться с помощью обычных команд, предназначенных для работы с памятью. В противном случае для чтения и записи регистров требуются специальные команды ввода-вывода, например `IN` и `OUT`. В обоих случаях ввод-вывод осуществляется путем передачи данных и информации о состоянии устройства между центральным процессором и указанными регистрами.

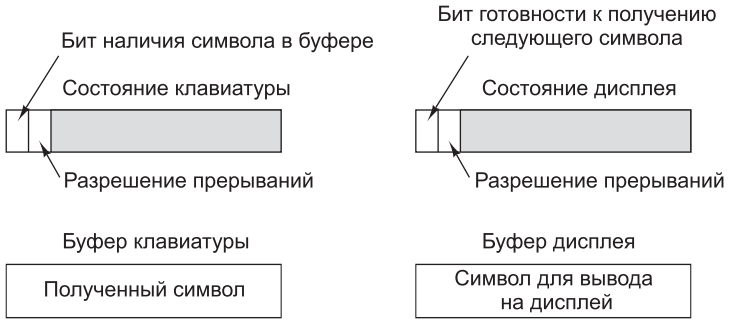


Рис. 5.19. Регистры устройств в простом терминале

В регистре состояния клавиатуры из восьми бит используются только два. Самый левый бит аппаратно устанавливается всякий раз, когда в буфере клавиатуры появляется символ. Если предварительно программно был установлен бит 6, выполняется прерывание. В противном случае прерывания не происходит (прерывания будут рассмотрены ниже). При программируемом вводе-выводе для получения входных данных центральный процессор обычно периодически в цикле считывает регистр состояния клавиатуры, пока бит 7 не получит значение 1. Когда это случается, программно считывается буферный регистр клавиатуры, чтобы получить символ. Считывание регистра данных вызывает сброс бита наличия символа.

Вывод осуществляется аналогично. Чтобы вывести символ на экран, сначала программно считывается регистр состояния дисплея, чтобы узнать, установлен ли бит готовности. Если он не установлен, цикл выполняется снова и снова до тех пор, пока бит готовности не станет равным единице. Это будет означать, что устройство готово принять символ. Как только терминал приходит в состояние готовности, символ программно записывается в буферный регистр дисплея, который выводит символ на экран и дает сигнал устройству сбросить бит готовности в регистре состояния дисплея. Когда символ появляется на экране, а терминал подготавливается к обработке следующего символа, контроллер снова устанавливает бит готовности.

В качестве примера программируемого ввода-вывода рассмотрим Java-процедуру (листинг 5.5). Эта процедура вызывается с двумя параметрами: массивом

символов, которые нужно вывести, и количеством символов, имеющимся в массиве (до килобита). Тело процедуры представляет собой цикл, в котором выводится по одному символу. Сначала центральный процессор ждет готовности устройства и только после этого выводит символ, и эта последовательность действий повторяется для каждого символа. Процедуры `in` и `out` — это типичные процедуры ассемблера для чтения и записи регистров устройств, которые определяются по первому параметру. Переменная, из которой выполняется чтение или в которую производится запись, определяется по второму параметру. Деление на 128 (путем сдвига вправо на 7 разрядов) устраняет младшие 7 бит, при этом бит готовности оказывается в нулевом разряде.

Листинг 5.5. Пример программируемого ввода-вывода

```
public static void output_buffer(int buf[], int count) {
  // Вывод блока данных на устройство
  int status, i, ready;

  for (i = 0; i<count; i++) {
    do {
      status = in(display_status_reg); // получение информации
                                       // о состоянии устройства
      ready = (status >> 7) & 0x01;    // выделение бита
                                       // готовности
    } while (ready == 1);
    out(display_buffer_reg, buf[i]);
  }
}
```

Основной недостаток программируемого ввода-вывода заключается в том, что центральный процессор проводит большую часть времени в цикле, ожидая готовности устройства. Этот процесс называется **активным ожиданием**. Если центральному процессору больше ничего не нужно делать (как, например, в стиральной машине), в этом нет ничего страшного (хотя даже простому контроллеру часто нужно контролировать несколько параллельных процессов). Но если процессору приходится выполнять еще какие-либо действия, например запускать другие программы, то активное ожидание здесь не подходит и нужно искать другие методы ввода-вывода.

Чтобы избавиться от активного ожидания, необходимо, чтобы центральный процессор запускал устройство ввода-вывода, а это устройство после завершения своей работы сообщало об этом процессору с помощью прерывания. Посмотрите на рис. 5.19. Установив бит разрешения прерываний в регистре устройства, программа говорит о том, что ждет от аппаратуры сигнала о завершении работы устройства ввода-вывода. Подробнее мы рассмотрим прерывания далее в этой главе, когда перейдем к вопросам передачи управления.

Во многих компьютерах сигнал прерывания порождается путем логического умножения (И) бита разрешения прерываний и бита готовности устройства. Если в первую очередь (перед запуском устройства ввода-вывода) программно разрешить прерывания, прерывание произойдет сразу же, поскольку бит готовности уже установлен. То есть вероятно сначала нужно запустить устройство, а уже после этого разрешить прерывания. Запись байта в регистр состояния устройства не меняет бита готовности, который может только считываться.

Ввод-вывод с управлением по прерываниям — это большой шаг вперед по сравнению с программируемым вводом-выводом, но все же он далеко не совершенен. Дело в том, что прерывание приходится генерировать для каждого передаваемого символа и нужно каким-то образом избавляться от слишком большого числа прерываний.

Решение лежит в возвращении к программируемому вводу-выводу, но только эту работу вместо центрального процессора должен делать кто-то другой. Посмотрите на рис. 5.20. Мы добавили новую микросхему — контроллер **прямого доступа к памяти (ПДП)**, имеющий непосредственный доступ к шине.

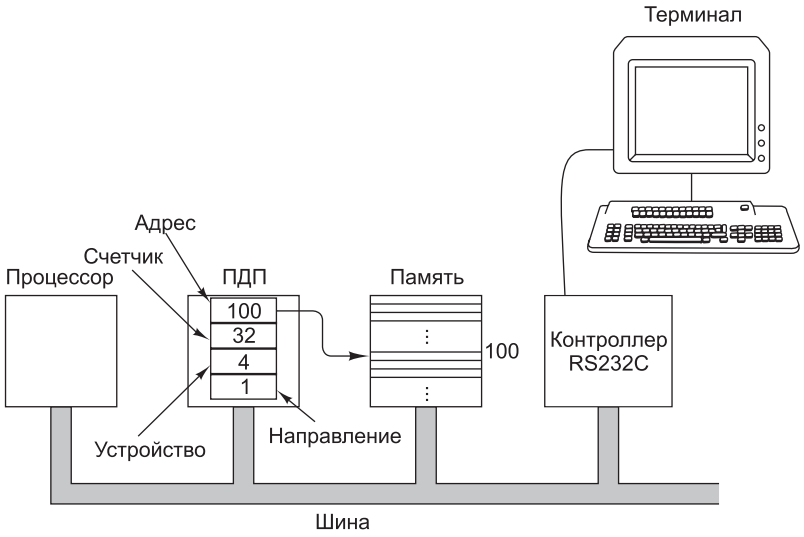


Рис. 5.20. Система с контроллером прямого доступа к памяти

Микросхема ПДП имеет по крайней мере 4 регистра. Все они могут загружаться программно центральным процессором. Первый регистр содержит адрес памяти, который нужно считать или записать. Второй регистр содержит количество передаваемых байтов или слов. Третий регистр содержит номер устройства ввода-вывода или адрес в адресном пространстве ввода-вывода, задавая требуемое устройство. Четвертый регистр сообщает, должны ли данные считываться с устройства или записываться на него.

Чтобы записать блок из 32 байт с адреса памяти 100 на терминал (например, устройство 4), центральный процессор записывает числа 32, 100 и 4 в первые три регистра ПДП, а код записи (например, 1) — в четвертый регистр, как показано на рис. 5.20. Контроллер ПДП, инициализированный таким образом, делает запрос на доступ к шине, чтобы считать байт 100 из памяти, точно так же, как если бы этот байт считывал центральный процессор. Получив нужный байт, контроллер ПДП посылает устройству 4 запрос на ввод-вывод, чтобы записать на него байт. После завершения этих двух операций контроллер ПДП увеличивает значение регистра адреса на 1 и уменьшает значение регистра счетчика на 1. Если значение счетчика остается положительным, следующий байт считывается из памяти и записывается на устройство ввода-вывода.

Когда значение счетчика доходит до 0, контроллер ПДП останавливает передачу данных и нагружает линию прерывания на микросхеме процессора. При наличии ПДП центральному процессору остается только инициализировать несколько регистров, после чего он может заниматься чем-нибудь другим, пока передача данных не завершится. При завершении передачи центральный процессор получает сигнал прерывания от контроллера ПДП. Некоторые контроллеры ПДП содержат 2, 3 и более наборов регистров, так что они могут управлять несколькими процессами передачи одновременно.

Отметим, что если какое-нибудь высокоскоростное устройство, например диск, будет запускаться контроллером ПДП, потребуется очень много циклов шины и для обращений к памяти, и для обращений к устройству. Во время этих циклов центральному процессору придется ждать (контроллер ПДП всегда имеет приоритет перед ЦП на доступ к шине, поскольку устройства ввода-вывода обычно не допускают задержек) — этот процесс называется **захватом цикла памяти**. Однако издержки, связанные с захватом циклов, не идут ни в какое сравнение с экономией, получаемой благодаря тому, что при передаче каждого байта (слова) не нужно обрабатывать прерывание.

Команды процессора Core i7

В этом и следующих двух подразделах мы рассмотрим наборы команд трех машин: Core i7, OMAP4430 ARM и ATmega168 AVR. Каждая из них поддерживает базовые команды, обычно получаемые в результате работы компиляторов, а также дополнительные команды, которые используются редко или используются исключительно операционной системой. Начнем с Core i7 — самого сложного набора команд. По сравнению с ним все остальные выглядят просто.

Команды Core i7 представляют собой причудливую смесь 32-разрядных команд, а также команд, появившихся еще в процессоре 8088. В табл. 5.9 приведены наиболее распространенные целочисленные команды, при этом используются следующие обозначения:

- SRC — источник данных;
- DST — приемник данных;
- # — количество битов, на которое происходит сдвиг;
- LV — количество локальных переменных.

Таблица 5.9. Наиболее распространенные целочисленные команды процессора Core i7

Команда	Описание
Команды перемещения	
MOV DST, SRC	Перемещение из SRC в DST
PUSH SRC	Помещение из SRC в стек
POP DST	Выталкивание слова из стека и помещение его в DST
XCHG DS1, DS2	Смена мест DS1 и DS2
LEA DST, SRC	Загрузка действительного адреса SRC в DST
CMOV DST, SRC	Условное перемещение

Команда	Описание
Арифметические команды	
ADD DST, SRC	Сложение SRC и DST
SUB DST, SRC	Вычитание SRC из DST
MUL SRC	Умножение EAX на SRC (без учета знака)
IMUL SRC	Умножение EAX на SRC (с учетом знака)
DIV SRC	Деление EDX:EAX на SRC (без учета знака)
IDV SRC	Деление EDX:EAX на SRC (с учетом знака)
ADC DST, SRC	Сложение SRC с DST и прибавление бита переноса
SBB DST, SRC	Вычитание DST и перенос из SRC
INC DST	Инкремент (прибавление 1) DST
DEC DST	Декремент (вычитание 1) DST
NEG DST	Отрицание DST (вычитание DST из 0)
Двоично-десятичные команды	
DAA	Десятичная коррекция
DAS	Десятичная коррекция для вычитания
AAA	Коррекция ASCII-кода для сложения
AAS	Коррекция ASCII-кода для вычитания
AAM	Коррекция ASCII-кода для умножения
AAD	Коррекция ASCII-кода для деления
Логические команды	
AND DST, SRC	Логическая операция И над SRC и DST
OR DST, SRC	Логическая операция ИЛИ над SRC и DST
XOR DST, SRC	Логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ над SRC и DST
NOT DST	Замещение DST дополнением до 1
Команды обычного и циклического сдвига	
SAL/SAR DST, #	Сдвиг DST влево/вправо на # бит
SHL/SHR DST, #	Логический сдвиг DST влево/вправо на # бит
ROL/ROR DST, #	Циклический сдвиг DST влево/вправо на # бит
RCL/RCR DST, #	Циклический сдвиг DST по переносу на # бит
Команды тестирования и сравнения	
TST SRC1, SRC2	Операнды логической операции И, установка флагов
CMP SRC1, SRC2	Установка флагов на основе разности SRC1 – SRC2
Команды передачи управления	
JMP ADDR	Переход к адресу
Jxx ADDR	Условные переходы на основе флагов

продолжение ⇩

Таблица 5.9 *(продолжение)*

Команда	Описание
CALL ADDR	Вызов процедуры по адресу
RET	Выход из процедуры
IRET	Выход из прерывания
LOOPxx	Продолжение цикла до выполнения определенного условия
INT ADDR	Программное прерывание
INTO	Прерывание, если установлен бит переполнения
Команды обработки строк	
LODS	Загрузка строки
STOS	Сохранение строки
MOVS	Перемещение строки
CMPS	Сравнение двух строк
SCAS	Сканирование строки
Команды для работы с кодами условий	
STC	Установка бита переноса в регистре EFLAGS
CLC	Сброс бита переноса в регистре EFLAGS
CMC	Дополнение бита переноса в регистре EFLAGS
STD	Установка бита направления в регистре EFLAGS
CLD	Сброс бита направления в регистре EFLAGS
STI	Установка бита прерывания в регистре EFLAGS
CLI	Сброс бита прерывания в регистре EFLAGS
PUSHFD	Помещение значения из регистра EFLAGS в стек
POPFD	Выталкивание значения из стека в регистр EFLAGS
LAHF	Загрузка AH из регистра EFLAGS
SAHF	Сохранение AH в регистре EFLAGS
Прочие команды	
SWAP DST	Изменение порядка следования байтов в DST
CWQ	Расширение EAX до EDX:EAX для деления
CWDE	Расширение 16-разрядного числа в AX до EAX
ENTER SIZE, LV	Создание стекового кадра с байтами размера
LEAVE	Удаление стекового кадра, созданного командой ENTER
NOP	Пустая операция
HLT	Останов
IN AL, PORT	Перенос байта из порта в АЛУ
OUT PORT, AL	Перенос байта из АЛУ в порт
WAIT	Ожидание прерывания

Перечень далеко не полный, поскольку в него не вошли команды с плавающей точкой, команды управления, а также некоторые не часто встречающиеся целочисленные команды (например, использование 8-разрядного байта для выполнения поиска по таблице). Тем не менее таблица дает представление о том, какие действия может выполнять Core i7.

Многие команды Core i7 обращаются к одному или к двум операндам, которые находятся в регистрах или памяти. Например, бинарная команда `ADD` складывает два операнда, а унарная команда `INC` увеличивает значение одного операнда на 1. Некоторые команды имеют несколько похожих вариантов. Например, команды сдвига могут сдвигать слово либо вправо, либо влево, с учетом знакового бита или без учета. Большинство команд имеют несколько различных кодировок в зависимости от природы операндов.

При выполнении команд источники данных (SRC) не изменяются, а приемники (DST) обычно изменяются. Существуют определенные правила, определяющие, что может быть источником, а что приемником; они несколько хаотично изменяются от команды к команде, но здесь мы не будем о них говорить. Многие команды имеют три варианта: для 8-, 16- и 32-разрядных операндов соответственно. Они различаются по коду операции и/или по одному биту в команде. В табл. 5.9 приведены в основном 32-разрядные команды.

Для удобства мы разделили команды на несколько групп. Первая группа содержит команды, которые перемещают данные между компонентами машины: регистрами, памятью и стеком. Вторая группа содержит арифметические команды для операций со знаком и без знака. При умножении и делении 64-разрядное произведение или делимое хранится в двух регистрах: EAX (младшие биты) и EDX (старшие биты).

Третья группа включает двоично-десятичную арифметику. Здесь каждый байт рассматривается как два 4-разрядных **полубайта**. Каждый полубайт содержит один десятичный разряд (от 0 до 9). Комбинации битов от 1010 до 1111 не используются. Таким образом, 16-разрядное целое число может содержать десятичное число от 0 до 9999. Хотя такая форма хранения неэффективна, она устраняет необходимость преобразования десятичных входных данных в двоичные, а затем обратно в десятичные для вывода. Эти команды служат для выполнения арифметических действий над двоично-десятичными числами. Они широко используются в программах на языке COBOL.

Логические команды и команды сдвига манипулируют битами в слове или байте в нескольких возможных комбинациях.

Следующие две группы связаны с проверкой, сравнением и осуществлением перехода в зависимости от полученного результата. Результаты проверки и сравнения хранятся в различных битах регистра EFLAGS. Символами J*xx* обозначена группа команд, выполняющих условный переход в зависимости от результатов предыдущего сравнения (то есть в зависимости от битов в регистре EFLAGS).

Core i7 поддерживает несколько команд для загрузки, сохранения, перемещения, сравнения и сканирования символьных строк или слов. Перед этими командами может стоять специальный префиксный байт `REP` (repetition — повторение), который заставляет команду повторяться до тех пор, пока не будет выполнено определенное условие (например, пока регистр ECX, значение которого уменьшается на 1 после каждого повторения, не станет равным 0). Таким образом, различные действия

(перемещение, сравнение и т. д.) могут производиться над произвольными блоками данных. Следующая группа команд управляет кодами условий.

В последнюю группу входят команды, которые не вошли ни в одну из предыдущих групп. Это команды перекодирования, управления кадром стека, ввода-вывода и остановки процессора.

Команды Core i7 имеют ряд **префиксов**. Один из них (`REP`) мы уже упомянули. Префикс — это специальный байт, который может ставиться практически перед любой командой (подобно `WIDE` в `IJVM`). Как уже отмечалось, префикс `REP` заставляет команду, идущую за ним, повторяться до тех пор, пока регистр ECX не примет значение 0. Префиксы `REPZ` и `REPNZ` заставляют команду выполняться снова и снова, пока код выполнения условия Z не примет значение 1 или 0 соответственно. Префикс `LOCK` резервирует шину для всей команды, чтобы можно было осуществлять межпроцессорную синхронизацию. Другие префиксы используются для того, чтобы команда работала в 16- или 32-разрядном формате. При этом меняется не только длина операндов, но и полностью переопределяются режимы адресации. Наконец, в Core i7 реализована сложная схема сегментации, в которой задействованы код, данные, стек и дополнительные сегменты (наследие 8088). Префиксы позволяют регламентировать применение тех или иных сегментов при обращении к памяти. К счастью, эта проблема для нас не слишком актуальна.

Команды OMAP4430

Почти все целочисленные команды ARM пользовательского режима, которые могут быть сгенерированы компилятором, перечислены в табл. 5.10, при этом используются следующие обозначения:

- S1 — регистр-источник;
- S2IMM — источник (регистр или непосредственно данные);
- S3 — регистр-источник (при использовании трех регистров);
- DST — регистр-приемник;
- DST1 — регистр-приемник (1 или 2);
- DST2 — регистр-приемник (1 или 2);
- ADDR — адрес памяти;
- IMM — непосредственное значение;
- REGLIST — список регистров;
- PSR — регистр состояния процессора;
- cc — условие перехода.

Таблица 5.10. Основные команды процессора OMAP4430

Команда	Описание
Команды загрузки	
LDRSB DST, ADDR	Загрузка байта со знаком (8 бит)
LDRB DST, ADDR	Загрузка байта без знака (8 бит)
LDRSH DST, ADDR	Загрузка полуслова со знаком (8 бит)

Команда	Описание
LDRH DST, ADDR	Загрузка полуслова без знака (16 бит)
LDR DST, ADDR	Загрузка слова (32 бита)
LDM S1, REGLIST	Загрузка нескольких слов
Команды сохранения	
STRB DST, ADDR	Сохранение байта (8 бит)
STRH DST, ADDR	Сохранение полуслова (16 бит)
STR DST, ADDR	Сохранение слова (32 бита)
STM SRC, REGLIST	Сохранение нескольких слов
Арифметические команды	
ADD DST, S1, S2IMM	Сложение
ADD DST, S1, S2IMM	Сложение с переносом
SUB DST, S1, S2IMM	Вычитание
SUB DST, S1, S2IMM	Вычитание с переносом
RSB DST, S1, S2IMM	Обратное вычитание
RSC DST, S1, S2IMM	Обратное вычитание с переносом
MUL DST, S1, S2	Умножение
MLA DST, S1, S2, S3	Умножение с накоплением
UMULL D1, D2, S1, S2	Длинное умножение без знака
SMULL D1, D2, S1, S2	Длинное умножение со знаком
UMLAL D1, D2, S1, S2	Длинное умножение с накоплением без знака
SMULL D1, D2, S1, S2	Длинное умножение с накоплением со знаком
CMP S1, S2IMM	Сравнение с установкой PSR
Команды обычного и циклического сдвига	
LSL DST, S1, S2IMM	Логический сдвиг влево
LSR DST, S1, S2IMM	Логический сдвиг влево
ASR DST, S1, S2IMM	Логический сдвиг вправо
ROR DSR, S1, S2IMM	Логический сдвиг вправо
Логические команды	
TST DST, S1, S2IMM	Проверка битов
TEQ DST, S1, S2IMM	Проверка равенства
AND DST, S1, S2IMM	Логическое И
EOR DST, S1, S2IMM	Логическое ИСКЛЮЧАЮЩЕЕ-ИЛИ
ORR DST, S1, S2IMM	Логическое ИЛИ
BIC DST, S1, S2IMM	Сброс битов

продолжение ⇩

Таблица 5.10 *(продолжение)*

Команда	Описание
Команды передачи управления	
Bcc IMM	Переход по адресу PC+IMM
BLcc IMM	Переход со связью по адресу PC+IMM
BLcc S1	Переход со связью по адресу, хранящемуся в регистре
Прочие команды	
MOV DST, S1	Перемещение
MOVT DST, IMM	Перемещение IMM в старшие биты
MVN DST, S1	Инвертирование регистра
MRS DST, PSR	Чтение PSR
MSR PSR, S1	Запись PSR
SWP DST, S1, ADDR	Перестановка регистра/слова памяти
SWPb DST, S1, ADDR	Перестановка регистра/байта памяти
SWI IMM	Программное прерывание

В таблице не представлены команды с плавающей точкой, команды управления (например, команды управления кэш-памятью, команды перезагрузки системы), команды, включающие отличные от пользовательских адресные пространства, устаревшие команды. Набор команд удивительно мал. Оно и понятно, ведь OMAP4430 — это RISC-процессор.

Структура команд `LDR` и `STR` очень проста. Эти команды имеют варианты для 1, 2 и 4 байт. Если в (32-разрядный) регистр загружается число размером меньше 32 бит, это число может быть либо расширено по знаку, либо дополнено нулями. Существуют команды для обоих вариантов.

Следующая группа команд предназначена для арифметических операций с возможной установкой битов в регистре состояния процессора. В CISC-машинах большинство команд устанавливают коды условия, но в RISC-машине это нежелательно, поскольку ограничивает способность компилятора сортировать команды, чтобы минимизировать задержки. Если изначальный порядок следования команд A ... B ... C, где A устанавливает коды условий, а B их проверяет, то компилятор не сможет вставить команду C между A и B, если тоже C устанавливает коды условий. По этой причине многие команды имеют два варианта, а компилятор обычно использует ту команду, которая не устанавливает коды условий, если не планируется проверить их позже. Чтобы задать установку кодов состояния, программист присоединяет к коду операции суффикс «S» — например, ADDS. Специальный бит в команде указывает процессору на необходимость установки кодов условия. Поддерживаются также команды умножения, деления с учетом знака и деления без учета знака.

Группа команд сдвига включает одну команду сдвига влево и две команды сдвига вправо. Каждая из них работает с 32-разрядными регистрами. Команды сдвига в основном используются для манипулирования битами. Циклический сдвиг применяется в основном при криптографических операциях и обработке

графики. Большинство CISC-машин имеют довольно много разнообразных команд обычного и циклического сдвига, но практически все они совершенно бесполезны. Никто из разработчиков компиляторов не будет по ним скорбеть.

Логические команды аналогичны арифметическим. Эта группа включает команды AND (И), EOR (ИСКЛЮЧАЮЩЕЕ ИЛИ), ORR (ИЛИ), TST, TEQ и BIC. Полезность последних трех команд сомнительна, но они могут выполняться за один цикл и не требуют практически никакого дополнительного аппаратного обеспечения, а потому часто включаются в набор команд. Даже разработчики RISC-машин порой поддаются искушению.

Следующая группа содержит команды передачи управления. Символами B*cc* обозначена группа команд, которые совершают переходы в зависимости от различных условий. Команды BL*cc* делают то же, но также помещают адрес следующей команды в регистр связи (R14). Они удобны для реализации вызовов процедур. В отличие от других RISC-архитектур, специальной команды для перехода по адресу в регистре не существует. Такая команда легко синтезируется из команды MOV, у которой в качестве приемника используется счетчик команд (R15).

Предусмотрено два способа вызова процедур. Первая команда BL*cc* использует формат перехода на рис. 5.10 с 24-разрядным смещением относительно PC. Это позволяет выполнить переход к любой команде, удаленной от текущей на 32 Мбайта в любом направлении. Вторая команда BL*cc* осуществляет переход по адресу в заданном регистре. Она может использоваться для реализации вызовов процедур с динамической компоновкой (например, виртуальных функций C++) или вызовов, смещение которых превышает 32 Мбайт.

В последней группе содержатся команды, не попавшие ни в одну из других групп. Команда MOVT введена из-за невозможности поместить 32-разрядный непосредственный операнд в регистр. Команда MOVT устанавливает биты с 16 по 31, а затем следующая команда передает оставшиеся биты, используя непосредственный формат. Команды MRS и MSR позволяют выполнять чтение и запись слова состояния процессора (PSR). Команды SWP выполняют атомарную перестановку содержимого регистра и слова памяти. Они реализуют примитивы многопроцессорной синхронизации, о которой будет рассказано в главе 8. Наконец, команда SWI инициирует программное прерывание — а проще говоря, вызывает системную функцию.

Команды ATmega168

Набор команд ATmega168 очень прост. Все они представлены в табл. 5.11. В каждой строке таблицы указан мнемонический код, краткое описание и фрагмент псевдокода, поясняющего действие команды. Вполне объяснимо обилие команд MOV для перемещения данных между регистрами. Также предусмотрены команды помещения элементов в стек и их выталкивания из стека. Указатель стека устанавливается в специальном 16-разрядном регистре (SP). Обращение к памяти может осуществляться по непосредственному адресу, посредством косвенной адресации через регистр или косвенной адресации через регистр со смещением. Чтобы сделать возможной адресацию до 64 Кбайт, команда загрузки с непосредственным адресом является 32-разрядной. Режимы косвенной адресации

используют пары регистров X, Y и Z, объединяющие два 8-разрядных регистра в один 16-разрядный.

Таблица 5.11. Набор команд процессора ATmega168

Команда	Описание	Семантика
ADD DST,SRC	Сложение	DST ← DST + SRC
ADC DST,SRC	Сложение с переносом	DST ← DST + SRC + C
ADIW DST,IMM	Непосредственное сложение со словом	DST+1:DST ← DST+1:DST + IMM
SUB DST,SRC	Вычитание	DST ← DST - SRC
SUBI DST,IMM	Непосредственное вычитание	DST ← DST - IMM
SBC DST,SRC	Вычитание с переносом	DST ← DST - SRC - C
SBCI DST,IMM	Непосредственное вычитание с переносом	DST ← DST - IMM — C
SBIW DST,IMM	Непосредственное вычитание со словом	DST+1:DST ← DST+1:DST - IMM
AND DST,SRC	Логическая операция ИЛИ	DST ← DST AND SRC
ANDI DST,IMM	Логическая операция ИЛИ с непосредственным операндом	DST ← DST AND IMM
OR DST,SRC	Логическая операция ИЛИ	DST ← DST OR SRC
ORI DST,IMM	Непосредственная логическая операция ИЛИ	DST ← DST OR IMM
EOR DST,SRC	Логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ	DST ← DST XOR SRC
COM DST	Поразрядное дополнение	DST ← 0xFF - DST
NEG DST	Дополнение до двух	DST ← 0x00 - DST
SBR DST,IMM	Установка битов в регистре	DST ← DST OR IMM
CBR DST,IMM	Сброс битов в регистре	DST ← DST AND (0xFF - IMM)
ING DST	Инкремент	DST ← DST + 1
DEC DST	Декремент	DST ← DST - 1
TST DST	Проверка на нуль или отрицательность	DST ← DST AND DST
CLR DST	Очистка регистра	DST ← DST XOR DST
SER DST	Установка регистра	DST ← 0xFF
MUL DST,SRC	Умножение без знака	R1:R0 ← DST * SRC
MULS DST,SRC	Умножение со знаком	R1:R0 ← DST * SRC
MULSU DST,SRC	Умножение со знаком и без	R1:R0 ← DST * SRC
RJMP IMM	Переход относительно PC	PC ← PC + IMM + 1

Команда	Описание	Семантика
IJMP	Косвенный переход по Z	PC ← Z (R30:R31)
JMP IMM	Переход	PC ← IMM
RCALL IMM	Относительный вызов	STACK ← PC+2, PC ← PC + IMM + 1
ICALL	Косвенный переход по Z	STACK ← PC+2, PC ← Z (R30:R31)
CALL	Вызов	STACK ← PC+2, PC ← IMM
RET	Возврат	PC ← STACK
CP DST,SRC	Сравнение	DST — SRC
CPC DST,SRC	Сравнение с переносом	DST — SRC — C
CPI DST,IMM	Сравнение с непосредственным операндом	DST — IMM
BRcc IMM	Переход по условию	if cc(true) PC ← PC + IMM + 1
MOV DST,SRC	Копирование регистра	DST ← SRC
MOVW DST,SRC	Копирование пары регистров	DST+1:DST ← SRC+1:SRC
LDI DST,IMM	Загрузка непосредственного значения	DST ← IMM
LDS DST,IMM	Прямая загрузка	DST ← MEM[IMM]
LD DST,XYZ	Косвенная загрузка	DST ← MEM[XYZ]
LDD DST,XYZ+IMM	Косвенная загрузка со смещением	DST ← MEM[XYZ+IMM]
STS IMM,SRC	Прямое сохранение	MEM[IMM] ← SRC
ST XYZ,SRC	Косвенное сохранение	MEM[XYZ] ← SRC
STD XYZ+IMM,SRC	Косвенное сохранение со смещением	MEM[XYZ+IMM] ← SRC
PUSH REGLIST	Занесение регистра в стек	STACK ← REGLIST
POP REGLIST	Извлечение регистра из стека	REGLIST ← STACK
LSL DST	Логический сдвиг влево на 1 разряд	DST ← DST LSL 1
LSR DST	Логический сдвиг вправо на 1 разряд	DST ← DST LSR 1
ROL DST	Циклический сдвиг влево на 1 разряд	DST ← DST ROL 1
ROR DST	Циклический сдвиг вправо на 1 разряд	DST ← DST ROR 1
ASR DST	Арифметический сдвиг вправо на 1 разряд	DST ← DST ASR 1

Сравнение наборов команд

Рассмотренные наборы команд разительно отличаются друг от друга. Core i7 — это классическая двухадресная 32-разрядная CISC-машина. Она пережила долгую историю, у нее особые и нерегулярные режимы адресации и многие ее команды обращаются непосредственно к памяти.

OMAP4430 — это современная трехадресная 32-разрядная RISC-машина с архитектурой загрузки/сохранения, минимальным набором режимов адресации, компактным и эффективным набором команд. Архитектура ATmega168 рассчитана на небольшой встроенный процессор, устанавливаемый на единственную микросхему.

Выбор всех трех архитектур не случаен. Набор команд компьютера Core i7 определяется тремя основополагающими факторами:

- обратная совместимость;
- обратная совместимость;
- обратная совместимость.

При нынешнем положении вещей никто не стал бы разрабатывать такую нерегулярную машину с таким небольшим количеством абсолютно разных регистров. По этой причине для Core i7 очень сложно писать компиляторы. Из-за недостатка регистров компиляторам постоянно приходится сохранять переменные в памяти, а затем вновь загружать их, что очень невыгодно даже при 3-уровневой кэш-памяти. Только благодаря таланту инженеров компании Intel процессор Core i7 работает достаточно быстро, несмотря на все недостатки его архитектуры команд. Но, как мы видели в главе 4, конструкция этого процессора чрезвычайно сложна.

Весьма современный уровень архитектуры набора команд представлен в процессоре OMAP4430. Это — полная 32-разрядная архитектура. Процессор содержит множество регистров, а в наборе команд преобладают 3-регистровые операции; имеется также небольшая группа команд загрузки и сохранения. Все команды одного размера, хотя число форматов совершенно невообразимо. Тем не менее реализация получается относительно прямолинейной и эффективной. Многие новые разработки очень похожи на OMAP4430, но форматов команд у них меньше.

В микросхеме ATmega168 реализован достаточно простой и стандартный набор команд, причем немного как самих команд, так и режимов адресации. Отличительные характеристики этого набора — 32 8-разрядных регистра для ускоренной обработки прерываний, возможность доступа к регистрам в пространстве памяти и на удивление мощные команды поразрядных операций. Основное преимущества такого решения состоит в том, что оно реализуется на незначительном числе транзисторов. Отсюда экономия пространства при размещении на кристалле, а значит, снижение стоимости процессора.

Поток управления

Потоком управления называют последовательность выполнения команд в ходе работы программы. При отсутствии переходов и вызовов процедур команды вызываются из последовательных ячеек памяти. Вызов процедуры влечет за

собой изменение потока управления, выполнение последовательности прерывается и начинается выполнение вызванной процедуры. Сопрограммы вызывают сходные изменения в потоке управления. Они нужны для моделирования параллельных процессов. Механизмы перехвата исключений и обработки прерываний тоже меняют поток управления при возникновении определенных ситуаций. Все это мы обсудим в следующих подразделах.

Последовательный поток управления и переходы

Большинство команд не меняют поток управления. После выполнения одной команды вызывается и выполняется та команда, которая расположена в памяти следом за выполненной. После выполнения каждой команды счетчик команд увеличивается на число, соответствующее длине команды. Значение счетчика команд представляет собой линейную функцию от времени — это значение растет на среднюю длину команды за средний промежуток времени. Иными словами, процессор выполняет команды в том же порядке, в котором они расположены в программе, как показано на рис. 5.21, *а*. Если программа содержит переходы, соответствие между порядком расположения команд в памяти и порядком их выполнения нарушается. При наличии переходов значение счетчика команд перестает быть монотонно возрастающей функцией от времени, как показано на рис. 5.21, *б*. В результате последовательность выполнения команд из самой программы уже не видна.

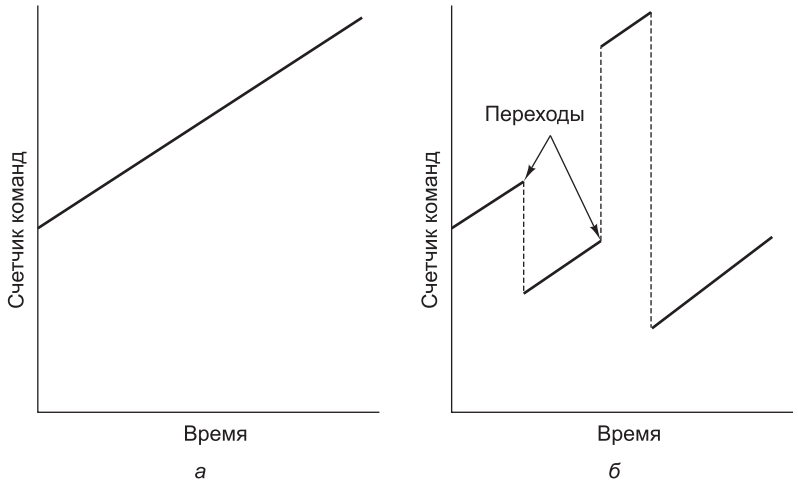


Рис. 5.21. Значение счетчика команд как функция от времени (приближенно): без переходов (*а*); с переходами (*б*)

Если программист не знает, в какой последовательности процессор будет выполнять команды, это может привести к ошибкам. Такое наблюдение вызвало к жизни известную статью Дейкстры о нежелательности применения оператора `goto` [1968a]. Эта статья привела к революции в программировании, одним из результатов которой стала замена `goto` операторами, позволяющих лучше структурировать поток управления, чем оператор `goto` — например, циклами `while`. Естественно, подобные программы в конце концов все равно компилируются

в программы уровня 2, содержащие многочисленные переходы, которых требует реализация операторов `if`, `while` и тому подобных высокоуровневых структур.

Процедуры

Самым важным инструментом структурирования программ является процедура. С одной стороны, вызов процедуры, как и команда перехода, изменяет поток управления, но в отличие от команды перехода, после выполнения процедуры управление возвращается команде, вызвавшей процедуру.

С другой стороны, тело процедуры можно рассматривать как определение новой команды на более высоком уровне. С этой точки зрения вызов процедуры можно считать самостоятельной командой, даже если процедура очень сложная. Чтобы понять часть программы, содержащую вызов процедуры, нужно знать, *что* и *как* эта процедура делает.

Особый интерес представляет **рекурсивная процедура**, которая вызывает саму себя либо непосредственно, либо через цепочку других процедур. Изучение рекурсивных процедур очень полезно для понимания того, как реализуются вызовы процедур и что в действительности представляют собой локальные переменные, поэтому давайте рассмотрим пример рекурсивной процедуры.

Ханойская башня — это название древней задачи, которая благодаря рекурсии имеет простое решение. В одном монастыре в Ханое есть три золотых колышка. На первый из них надето 64 концентрических золотых диска. Диаметр дисков уменьшается снизу вверх. Второй и третий колышки пусты. Монахи должны по одному перенести все диски на третий колышек, используя при необходимости второй колышек, но при этом после каждой итерации ни на одном из колышков диск большего диаметра не может оказаться на диске меньшего диаметра. Говорят, что когда они закончат, наступит конец света. Если вы хотите потренироваться, можете использовать пластиковые диски, и не 64, а поменьше. (Будем надеяться, что когда вы решите эту задачу, ничего страшного не произойдет. Чтобы произошел конец света, требуются именно 64 диска, причем обязательно золотых.) На рис. 5.22 показана начальная конфигурация для случая, когда число дисков (n) равно 5.

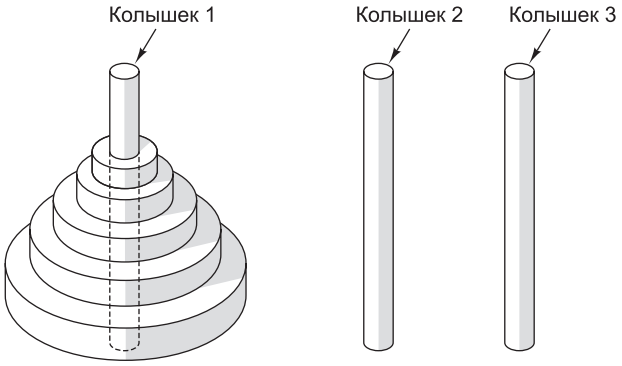


Рис. 5.22. Исходное положение дисков в задаче «Ханойская башня» для пяти дисков

Чтобы переместить n дисков с колышка 1 на колышек 3, нужно сначала перенести $n - 1$ дисков с колышка 1 на колышек 2, затем перенести один диск с колышка 1 на колышек 3, потом перенести $n - 1$ диск с колышка 2 на колышек 3. Решение этой задачи для трех дисков иллюстрирует рис. 5.23.

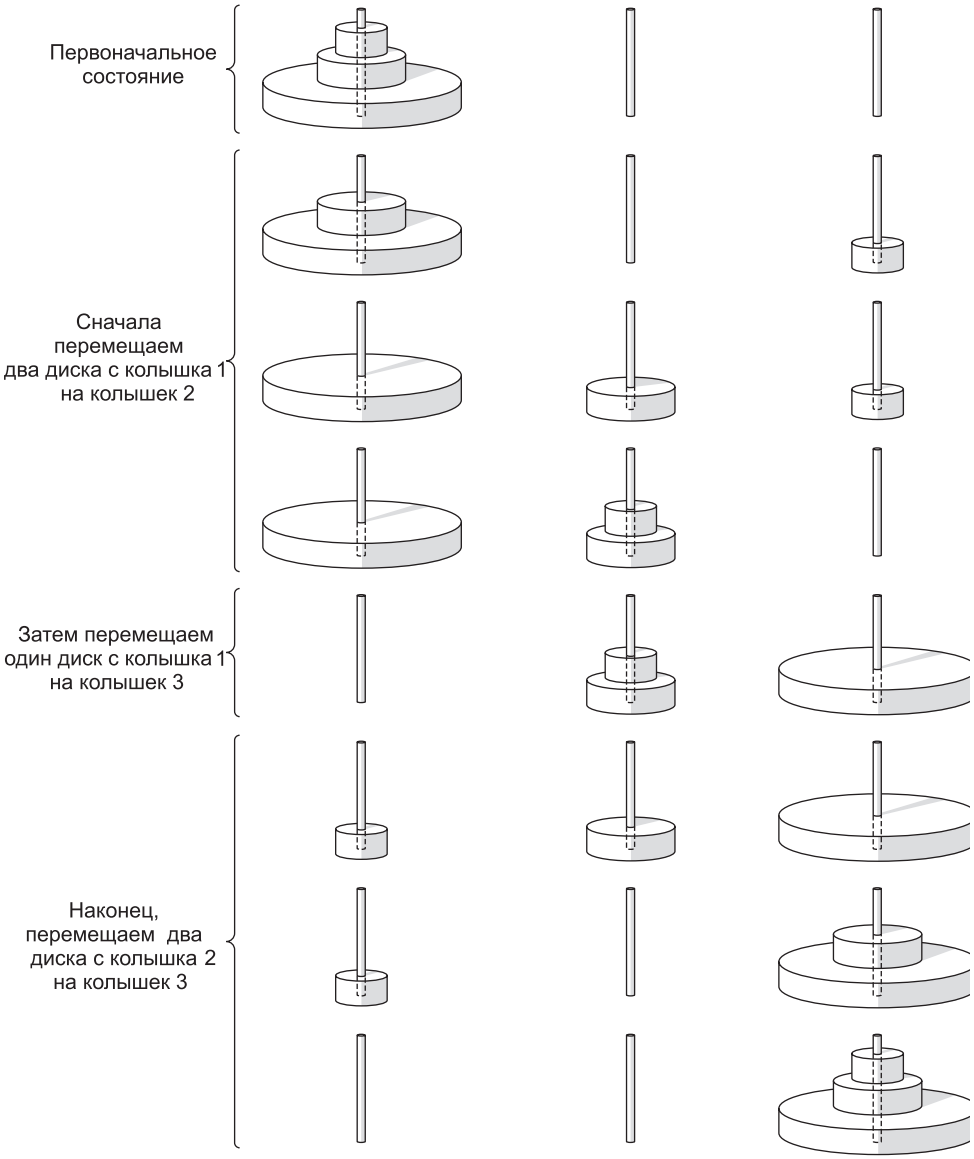


Рис. 5.23. Решение задачи «Ханойская башня» для трех дисков

Для решения задачи нам нужна процедура, которая позволяет переместить `n` дисков с колышка `i` на колышек `j`:

```
towers (n, i, j)
```

После вызова этой процедуры решение должно выводиться на экран. Сначала процедура проверяет, равно ли единице значение `n`. Если да, то решение тривиально: нужно просто переместить один диск с `i` на `j`. Если `n` не равно 1, решение состоит из трех частей и каждая из этих частей представляет собой рекурсивную процедуру.

Все решение представлено в листинге 5.6. Рассмотрим такой вызов процедуры:

```
towers (3, 1, 3)
```

Этот вызов порождает еще три вызова:

```
towers (2, 1, 2)
towers (1, 1, 3)
towers (2, 2, 3)
```

Первый и третий вызов производят по три вызова каждый, и всего получится семь.

Листинг 5.6. Процедура для решения задачи «Ханойская башня»

```
public void towers (int n, int i, int j) {
  int k;
  if (n == 1)
    System.out.println("Переместить диск с " + i + "на" + j);
  else {
    k=6-i-j;
    towers(n-1, i, k);
    towers (1, i, j);
    towers (n-1, k, j);
  }
}
```

Для рекурсивных процедур нам нужен стек, чтобы, как и в IJVM, хранить параметры и локальные переменные каждого вызова. Каждый раз при вызове процедуры на вершине стека располагается новый стековый кадр для процедуры. Текущий кадр — это кадр, созданный последним. В наших примерах стек растет снизу вверх, от малых адресов к большим, как и в IJVM.

Помимо указателя стека (Stack Pointer, SP), который указывает на вершину стека, удобно иметь указатель кадра (Frame Pointer, FP), указывающий на заданное место в кадре, например на связующий указатель, как в IJVM, или на первую локальную переменную. На рис. 5.24 изображен стековый кадр для машины с 32-разрядным словом. При первом вызове процедуры `towers` в стек помещаются значения `n`, `i` и `j`, а затем выполняется команда `CALL`, которая помещает в стек адрес возврата, 1012. Вызванная процедура сохраняет в стеке старое значение FP (1000) в ячейке 1016, а затем передвигает указатель стека для обозначения места хранения локальных переменных. При наличии только одной 32-разрядной локальной переменной (`k`) указатель стека увеличивается на 4 — до 1020. На рис. 5.24, *а* показан результат всех этих действий.

Первое, что должна сделать процедура после вызова, — сохранить предыдущее значение FP (чтобы его можно было восстановить при выходе из процедуры), скопировать значение SP в FP и, возможно, увеличить SP на одно слово, в зависимости от того, куда указывает FP нового кадра. В этом примере FP указывает на первую локальную переменную (хотя в IJVM регистр LV указывал на

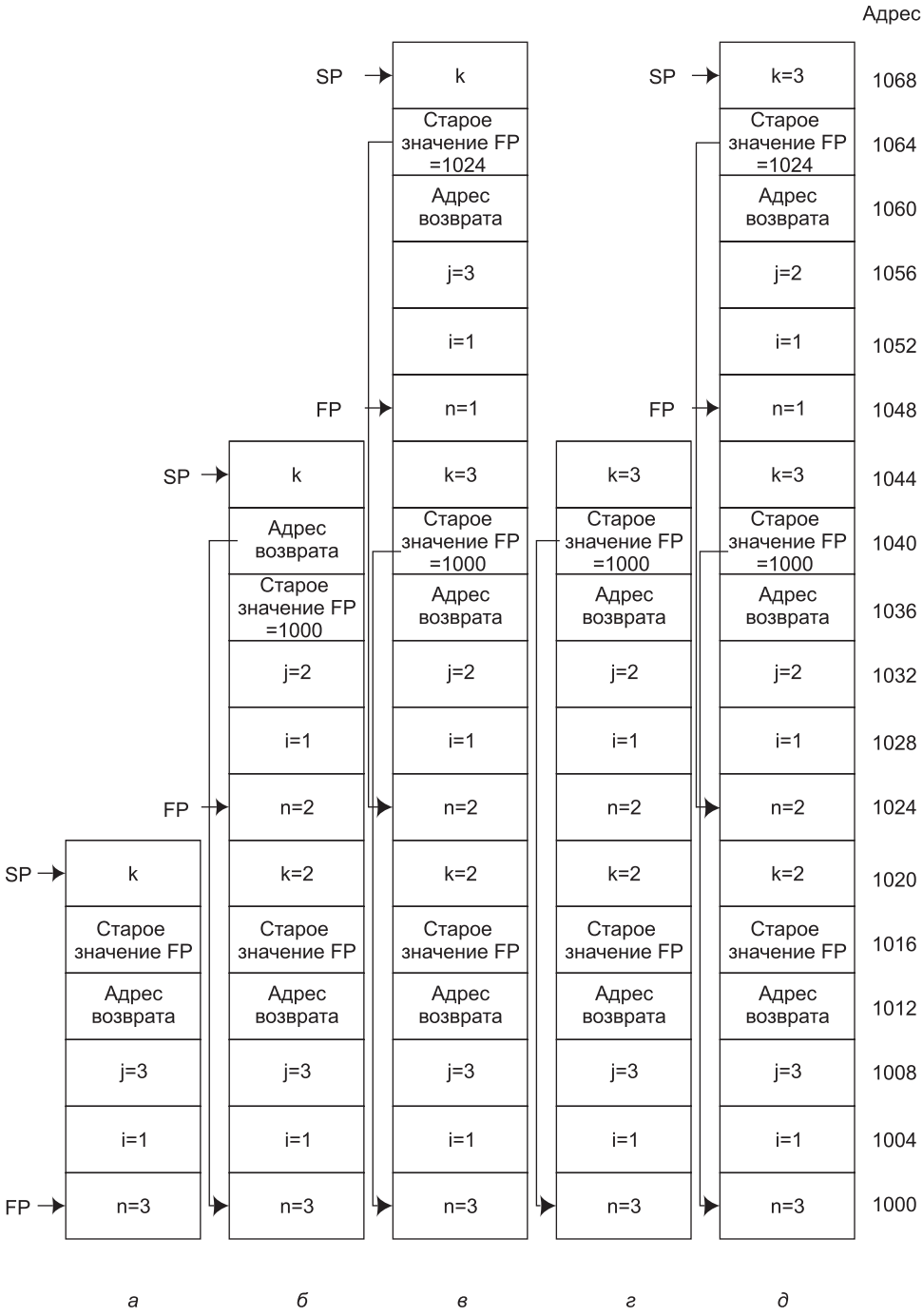


Рис. 5.24. Состояние стека во время выполнения программы из листинга 5.6

связующий указатель). Разные машины оперируют указателем кадра немного по-разному, иногда помещая его в самый низ стекового кадра, иногда в вершину, а иногда в середину, как на рис. 5.24. В этом отношении стоит сравнить рис. 5.24 с рис. 4.10, чтобы познакомиться с двумя разными способами обращения со связующим указателем. Возможны и другие способы. Но в любом случае обязательно должна быть возможность выйти из процедуры и восстановить предыдущее состояние стека.

Код, который сохраняет старый указатель кадра, устанавливает новый указатель кадра и увеличивает указатель стека, чтобы зарезервировать пространство для локальных переменных, называется **прологом процедуры**. При выходе из процедуры стек должен быть очищен, и эта задача решается в **эпилоге процедуры**. Одна из важнейших характеристик компьютера — насколько быстро он может выполнять пролог и эпилог. Если они очень длинные и выполняются медленно, вызывать процедуры оказывается невыгодно. Команды `ENTER` и `LEAVE` в Core i7 были разработаны специально для того, чтобы эффективно работали прологи и эпилоги процедур. Конечно, они поддерживают определенную модель обращения с указателем кадра, и если компилятор поддерживает другую модель, использовать эти команды нельзя.

А теперь вернемся к Ханойской башне. Каждый вызов процедуры добавляет новый кадр в стек, а каждый выход из процедуры удаляет кадр из стека. Посмотрим, как используется стек при реализации рекурсивных процедур, и начнем с вызова

```
towers (3, 1, 3)
```

На рис. 5.24, *а* показано состояние стека сразу после вызова процедуры. Сначала процедура проверяет, равно ли `n` единице, а установив, что `n` = 3, заполняет `k` и совершает вызов

```
towers (2, 1, 2)
```

Состояние стека после завершения этого вызова показано на рис. 5.24, *б*. Далее процедура выполняется снова (вызванная процедура всегда начинается с начала). На этот раз условие `n` = 1 снова не подтверждается, поэтому процедура снова заполняет `k` и совершает вызов

```
towers (1, 1, 3)
```

Состояние стека после этого вызова показано на рис. 5.24, *в*. Счетчик команд указывает на начало процедуры. На этот раз условие подтверждается, и на экран выводится строка. Затем совершается выход из процедуры. Для этого удаляется один кадр, а значения FP и SP переопределяются (рис. 5.24, *г*). Далее продолжается выполнение процедуры с адреса возврата:

```
towers (1, 1, 2)
```

Этот вызов добавляет новый кадр в стек (рис. 5.24, *д*). Печатается еще одна строка. После выхода из процедуры кадр удаляется из стека. Вызовы процедур продолжаются до тех пор, пока не завершится выполнение первой процедуры и пока кадр, изображенный на рис. 5.24, *а*, не будет удален из стека. Чтобы лучше понять, как работает рекурсия, нужно, используя только ручку и бумагу, полностью воспроизвести выполнение процедуры

```
towers (3, 1, 3)
```

Сопрограммы

В обычной последовательности вызовов между вызывающей и вызываемой процедурами есть очевидное различие. Рассмотрим процедуру A, которая вызывает процедуру B (рис. 5.25).

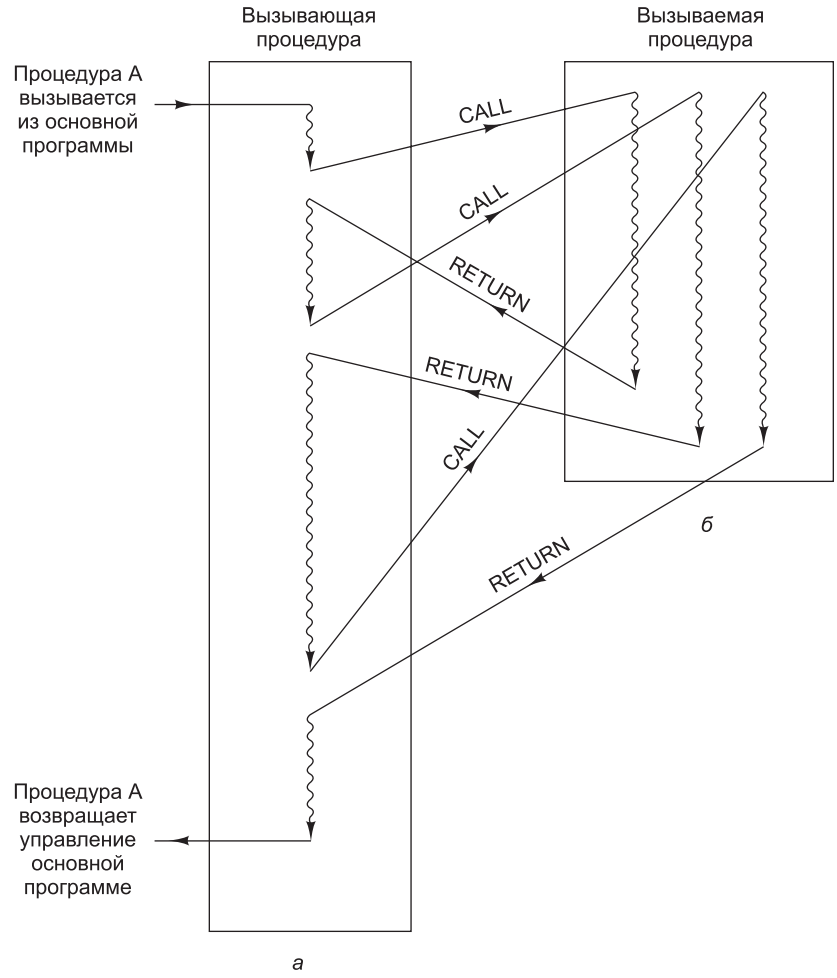


Рис. 5.25. Выполнение вызванной процедуры всегда начинается с ее начала

Процедура B работает какое-то время, затем возвращает управление A. На первый взгляд может показаться, что эти ситуации симметричны, поскольку ни A, ни B не являются главными программами — это процедуры (впрочем, процедура A может быть вызвана основной программой, но в данном случае это несущественно). Более того, сначала управление передается от A к B (при вызове), а затем — от B к A (при возвращении).

Отличие состоит в том, что когда управление переходит от A к B, процедура B начинает выполняться с самого начала; а при передачи управления из B обратно в A выполнение процедуры A продолжается не с начала, а с команды,

следующей за вызовом процедуры B. Если A работает некоторое время, а потом снова вызывает процедуру B, выполнение B снова начинается с самого начала, а не с того места, после которого управление было возвращено процедуре A. Если в процессе выполнения процедура A вызывает процедуру B многократно, процедура B каждый раз начинается с начала, а процедура A уже никогда больше с начала не начинается.

Это отличие отражается в способе передачи управления между A и B. Когда процедура A вызывает B, она использует команду вызова процедуры, которая помещает адрес возврата (то есть адрес той команды, которая в программе располагается следом за процедурой) в такое место, откуда его потом легко будет извлечь, например на вершину стека. Затем она помещает адрес процедуры B в счетчик команд, чтобы завершить вызов. Для выхода из процедуры B используется не команда вызова процедуры, а команда выхода из процедуры, которая просто выталкивает адрес возврата из стека и помещает его в счетчик команд.

Однако иногда нужно, чтобы обе процедуры (A и B) вызывали друг друга в качестве процедуры, как показано на рис. 5.26. При возврате из B в A процедура B совершает переход к тому оператору, перед которым последовал вызов

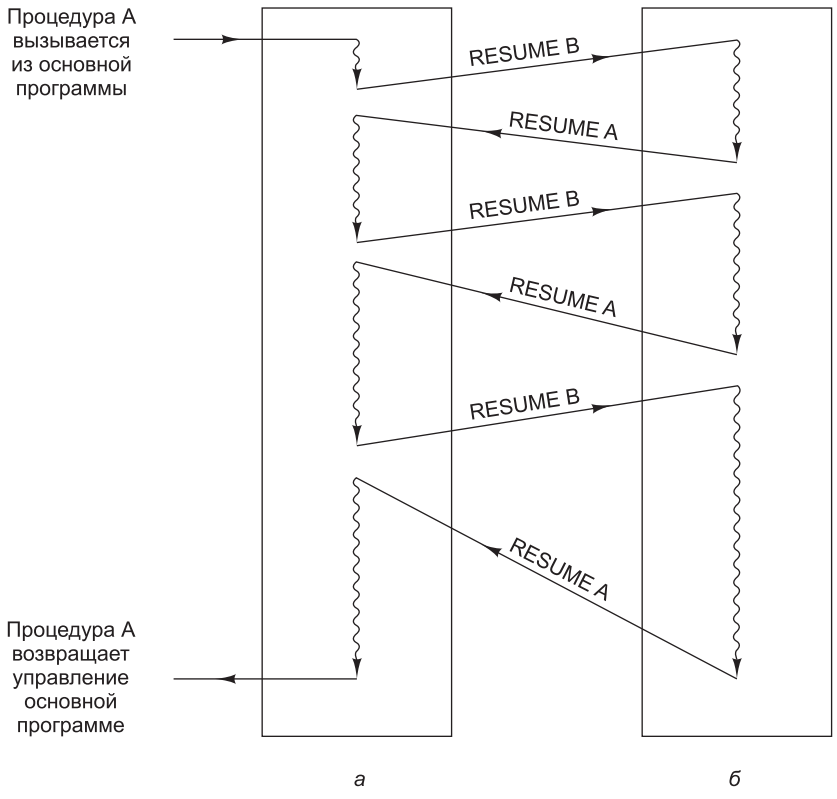


Рис. 5.26. После завершения сопрограммы выполнение начинается с того места, на котором оно завершилось в прошлый раз, а не с самого начала

процедуры B. Когда процедура A передает управление процедуре B, она возвращается не к самому началу B (за исключением первого раза), а к тому месту, перед котором произошел предыдущий вызов A. Две процедуры, работающие подобным образом, называются **сопрограммами**.

Сопрограммы обычно используются для параллельной обработке данных на одном процессоре. Каждая сопрограмма работает как бы одновременно с другими сопрограммами, как будто у нее есть собственный процессор. Такой подход упрощает программирование некоторых приложений. Он также полезен для проверки программного обеспечения, предназначенного для многопроцессорного исполнения.

Обычные команды `CALL` и `RETURN` для вызова сопрограмм не подходят, поскольку, хотя адрес перехода берется из стека, как и при возвращении управления, но в отличие от возвращения управления, при вызове сопрограммы адрес возврата помещается в определенном месте, чтобы в последующем к нему вернуться. Было бы неплохо, если бы существовала команда, в которой вместо вершины стека используется счетчик команд. Эта команда сначала должна выталкивать старый адрес возврата из стека и помещать его во внутренний регистр, затем помещать счетчик команд в стек и, наконец, копировать содержание внутреннего регистра в счетчик команд. Поскольку одно слово выталкивается из стека, а другое помещается в стек, состояние указателя стека не меняется. Такая команда встречается очень редко, поэтому в большинстве случаев ее приходится моделировать из нескольких команд.

Перехват исключений

Перехват исключений — это особый тип вызова процедур, который происходит при определенных условиях, обычно очень серьезных, но редко встречающихся. Один из примеров такого условия — переполнение. В большинстве процессоров, если результат выполнения арифметической операции превышает самое большое допустимое число, происходит исключение, которое перехватывается. Это значит, что поток управления переходит в какую-то фиксированную ячейку памяти, а не продолжается последовательно дальше. В этой фиксированной ячейке находится команда перехода к специальной процедуре (**обработчику исключений**), которая выполняет какое-либо определенное действие, например печатает сообщение об ошибке. Если результат операции находится в пределах допустимого, исключения не происходит.

Важно то, что исключения могут вызываться программно, а перехватываются они аппаратно или на микропрограммном уровне. Помимо перехвата исключения есть и другой способ определения факта переполнения. Для этого нужно иметь 1-разрядный регистр, который будет устанавливаться всякий раз, когда происходит переполнение. В этом случае программисту, который хотел бы проверять результат на переполнение, после каждой арифметической команды пришлось бы включать в программу команду перехода по переполнению, что очень неудобно. То есть по сравнению с явной программной проверкой перехват исключений экономит время и память.

Перехват исключений можно реализовать не только аппаратно, но и с помощью микропрограммы путем той же явной проверки. В этом случае при

обнаружении факта переполнения адрес обработчика исключений загружается в счетчик команд. Проверка на уровне микропрограммы требует меньше времени, чем проверка на уровне программы, поскольку может выполняться одновременно с каким-либо другим действием. Кроме того, такая проверка экономит память, поскольку ее можно реализовать только в одном месте, например в основном цикле микропрограммы независимо от того, сколько арифметических команд имеется в основной программе.

К наиболее распространенным условиям, которые могут вызывать исключения, относятся: переполнение и исчезновение значащих разрядов при выполнении операций с плавающей точкой, переполнение при выполнении операций с целыми числами, нарушения защиты, неопределяемый код операции, переполнение стека, запуск несуществующего устройства ввода-вывода, попытка выборки слова с нечетным адресом, деление на 0.

Прерывания

Прерывания — это изменения в потоке управления, вызванные не самой программой, а чем-либо другим. Обычно прерывания связаны с процессом ввода-вывода. Например, программа может дать команду диску начать передачу информации и инициировать прерывание, как только передача данных завершится. Как и в случае исключений, при прерываниях работа программы останавливается, и управление передается **программе обработки прерываний** (Interrupt Service Routine, **ISR**), или **обработчику прерываний**, который выполняет определенные действия. После завершения этих действий обработчик прерываний передает управление прерванной программе. Она должна заново начать прерванный процесс в том же самом состоянии, в котором находилась в момент прерывания. Это значит, что прежнее состояние всех внутренних регистров (то есть состояние, которое было до прерывания) должно быть восстановлено.

Различие между исключениями и прерываниями заключается в том, что исключения синхронны по отношению к программе, а прерывания асинхронны. Если многократно перезапускать программу с одними и теми же входными данными, исключения каждый раз будут происходить в одних и тех же местах программы, а прерывания — нет (в нашем примере с диском прерывание произойдет только тогда, когда диск завершит передачу данных, а не когда этого потребует программа). Причина воспроизводимости исключений и невоспроизводимости прерываний состоит в том, что первые вызываются программой непосредственно, а вторые — опосредовано.

Чтобы понять, как происходят прерывания, рассмотрим обычный пример: компьютеру нужно вывести на терминал строку символов. Программа сначала собирает в буфер все символы, предназначенные для вывода на экран, инициализирует глобальную переменную `ptr`, которая должна указывать на начало буфера, и делает вторую глобальную переменную `count` равной числу символов, выводимых на экран. Затем программа проверяет, готов ли терминал, и если готов, выводит на экран первый символ (например, используя регистры, показанные на рис. 5.19). Начав процесс ввода-вывода, центральный процессор освобождается и может запустить другую программу или сделать что-либо еще.

Через некоторое время символ отображается на экране. После этого может быть инициировано прерывание. Ниже перечислены основные шаги (в упрощенной форме).

Действия аппаратного обеспечения:

1. Контроллер устройства нагружает линию прерывания на системной шине.
2. Когда центральный процессор оказывается готовым к обработке прерывания, он выставляет на шине символ подтверждения прерывания.
3. Когда контроллер устройства обнаруживает, что сигнал прерывания подтвержден, он помещает небольшое целое число на информационные линии, чтобы «представиться» (то есть показать, что за устройство является источником прерывания). Это число называется **вектором прерываний**[1].
4. Центральный процессор считывает вектор прерывания с шины и временно его сохраняет.
5. Центральный процессор помещает в стек счетчик команд и слово состояния программы.
6. Центральный процессор определяет местонахождение нового счетчика команд, используя вектор прерывания в качестве индекса в таблице в нижней части памяти. Если, например, размер счетчика команд составляет 4 байта, тогда вектор прерываний *n* соответствует адресу 4*n*. Новый счетчик команд указывает на начало программы обработки прерываний для устройства, ставшего источником прерывания. Часто помимо этого загружается или изменяется слово состояния программы (например, чтобы блокировать дальнейшие прерывания).

Дальнейшие действия выполняются программно:

7. Программа обработки прерываний сохраняет все нужные ей регистры таким образом, чтобы их можно было восстановить позднее. Их можно сохранить в стеке или в системной таблице.
8. Каждый вектор прерывания используется всеми устройствами данного типа совместно, поэтому в данный момент еще не известно, какой терминал вызвал прерывание. Номер терминала можно узнать, считав значение какого-нибудь регистра.
9. После этого можно считывать любую другую информацию прерывания, например коды состояния.
10. Если происходит ошибка ввода-вывода, на этом этапе ее нужно обработать.
11. Глобальные переменные `ptr` и `count` обновляются. Первая увеличивается на 1, чтобы показывать на следующий байт, а вторая уменьшается на 1, чтобы указать, что осталось вывести на 1 байт меньше. Если `count` все еще больше 0, значит, еще не все символы выведены на экран. Тот символ, на который в данный момент указывает `ptr`, копируется в выходной буферный регистр.

[1] Автор не совсем прав: здесь речь должна идти о номере прерывания. Каждому типу прерывания соответствует свой номер. Термин «вектор прерываний» используется в случае, когда по номеру прерывания находится адрес программы обработки прерывания, и этот адрес представляется не одним значением, а несколькими, то есть приходится инициализировать более одного регистра. Другими словами, адрес представляется не скалярной величиной, а многомерной, векторной. — *Примеч. науч. ред.*

12. В случае необходимости выдается специальный код, который сообщает устройству или контроллеру прерывания, что прерывание обработано.
13. Восстанавливаются все сохраненные регистры.
14. Выполняется команда выхода из прерывания, возвращающая центральный процессор в то состояние, в котором он находился до прерывания. После этого компьютер продолжает работу с того места, в котором ее приостановил.

С прерываниями связано важное понятие **прозрачности**. Когда происходит прерывание, могут производиться разнообразные действия и запускаться разного рода программы, но когда все заканчивается, компьютер должен вернуться точно в то же состояние, в котором он находился до прерывания. Программа обработки прерываний, обладающая таким свойством, называется прозрачной.

Если компьютер имеет только одно устройство ввода-вывода, тогда прерывания работают именно так, как мы только что описали. Однако большой компьютер может содержать много устройств ввода-вывода, причем несколько устройств могут работать одновременно, возможно, у разных пользователей. Существует некоторая вероятность, что во время работы обработчика прерывания другое устройство ввода-вывода тоже попытается вызвать *свое* прерывание.

Здесь существует два подхода. Первый — для всех программ обработки прерываний сначала (даже до сохранения регистров) предотвратить последующие прерывания — в этом случае прерывания будут происходить строго поочередно. Однако это может привести к проблемам с устройствами, которые не могут долго простаивать. Например, на линию связи, поддерживающую скорость передачи 9600 бит в секунду, символы поступают каждые 1042 микросекунды. Если первый символ окажется необработанным, когда поступит второй, данные будут потеряны.

Если компьютер имеет подобные устройства ввода-вывода, то лучше всего приписать каждому устройству определенный приоритет, высокий — для более критичных, низкий — для менее критичных устройств. Центральный процессор тоже должен иметь приоритеты, которые определяются по одному из полей слова состояния программы. Если устройство с приоритетом n вызывает прерывание, программа обработки прерывания тоже должна работать с приоритетом n.

Если программа обработки прерывания выполняется с приоритетом n, любая попытка обработать прерывание от другого устройства с более низким приоритетом будет игнорироваться, пока программа обработки прерывания не завершится и центральный процессор не начнет выполнение программы более низкого приоритета. В то же время прерывания от устройств с более высоким приоритетом должны обрабатываться без задержек.

Поскольку сами программы обработки прерываний могут прерываться, единственно возможный способ точно управлять ситуацией — сделать так, чтобы все прерывания были прозрачными. Рассмотрим простой пример с несколькими прерываниями. Пусть компьютер имеет три устройства ввода-вывода: принтер, диск и линию RS232 с приоритетами 2, 4 и 5 соответственно. Изначально ($t = 0$, где t — время) работает пользовательская программа. При $t = 10$ принтер неожиданно инициирует прерывание. Запускается программа обработки прерывания (ISR) от принтера, как показано на рис. 5.27.

При $t = 15$ прерывания требует линия RS232. Так как линия RS232 имеет более высокий приоритет (5), чем принтер (2), инициируется обработка этого прерыва-

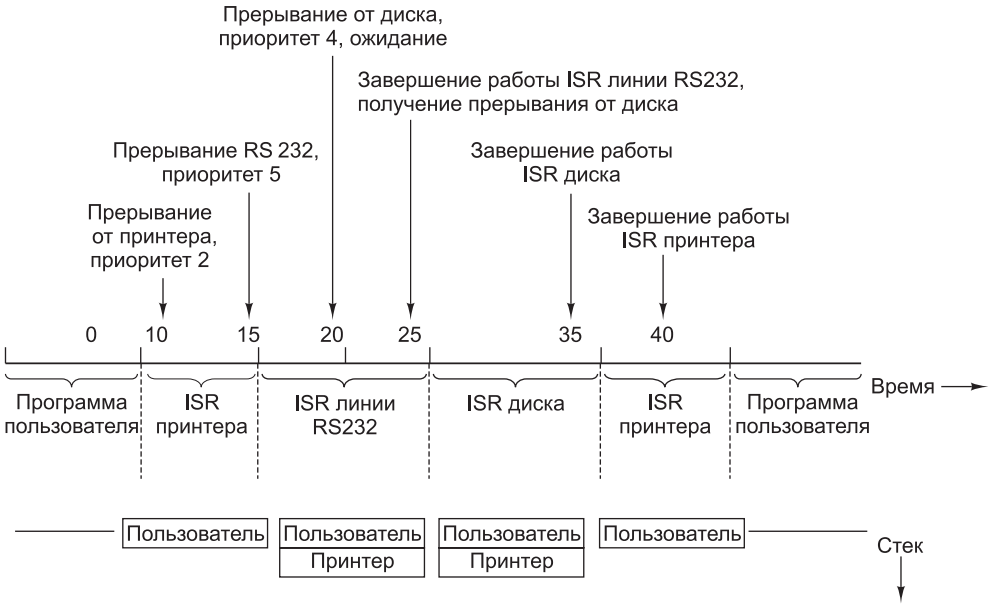


Рис. 5.27. Пример с несколькими прерываниями. Последовательность действий

ния. Состояние машины, при котором работает ISR принтера, сохраняется в стеке, и начинается выполнение программы обработки прерывания линии RS232.

Немного позже, при $t = 20$, диск завершает свою работу и сигнализирует об этом прерыванием. Однако его приоритет (4) ниже, чем приоритет работающей в данный момент программы обработки прерываний (5), поэтому центральный процессор не подтверждает прием сигнала прерывания, и диск вынужден ждать. При $t = 25$ заканчивает работать ISR линии RS232, и машина возвращается в то состояние, в котором она находилась до прерывания от линии RS232, то есть в состояние, соответствующее работе ISR принтера с приоритетом 2. Как только центральный процессор переключается на приоритет 2, еще до выполнения первой команды, диск с приоритетом 4 совершает прерывание, и запускается ISR диска. После ее завершения снова продолжается программа обработки прерываний от принтера. Наконец, при $t = 40$ все программы обработки прерываний завершаются, и выполнение пользовательской программы начинается с того места, на котором она прервалась.

Со времен процессора 8088 все процессоры Intel имеют два уровня (приоритета) прерываний: маскируемые и немаскируемые прерывания. Немаскируемые прерывания обычно используются только для сообщения об очень серьезных ситуациях, например об ошибках четности в памяти. Для всех устройств ввода-вывода имеется единственное маскируемое прерывание.

Когда устройство ввода-вывода требует прерывания, центральный процессор использует вектор прерывания при индексировании таблицы из 256 элементов, чтобы найти адрес программы обработки прерываний. Элементы таблицы представляют собой 8-байтные дескрипторы сегмента. Таблица может начинаться в любом месте памяти. Глобальный регистр указывает на ее начало.

При наличии только одного уровня прерываний у центрального процессора нет возможности сделать так, чтобы высокоприоритетное устройство прерывало работу среднеприоритетной программы обработки прерываний, пока этому мешает низкоприоритетное устройство. Для решения проблемы центральные процессоры Intel обычно используют внешний контроллер прерываний (например, 8259A). При первом прерывании (например, с приоритетом *n*) работа процессора приостанавливается. Если после этого происходит еще одно прерывание с более высоким приоритетом, контроллер прерывания инициирует прерывание во второй раз. Если же второе прерывание обладает более низким приоритетом, оно не инициируется до окончания первого. Чтобы эта система работала, контроллер прерываний должен каким-либо образом узнавать о завершении текущей программы обработки прерываний. Поэтому когда полностью завершается обработка текущего прерывания, центральный процессор должен посылать контроллеру прерываний специальную команду.

Ханойская башня

Теперь, когда мы изучили уровень архитектуры набора команд трех машин, нам нужно все обобщить. Давайте подробно рассмотрим все тот же пример решения задачи «Ханойская башня»). В листинге 5.6 приведена версия этой программы на языке Java.

Однако чтобы избежать проблем с вводом-выводом Java, для машин Core i7 и OMAP4430 мы будем транслировать версию программы не на Java, а на C. Единственное отличие — это замена Java-оператора `println` стандартным оператором языка C:

```
printf("Переместить диск с %d на %d\n", i, j)
```

Синтаксис строки в операторе `printf` не важен (строка печатается буквально за исключением символов `%d`, означающих, что следующее целое число будет представлено в десятичной системе счисления). Здесь важно только то, что процедура вызывается с тремя параметрами: форматирующей строкой и двумя целыми числами.

Мы использовали язык C для Core i7 и OMAP4430, поскольку библиотека ввода-вывода Java для этих машин недоступна, в отличие от библиотеки C ввода-вывода. Разница минимальна — всего один оператор вывода строки на экран.

Решение задачи «Ханойская башня» на ассемблере Core i7

В листинге 5.7 приведен возможный результат трансляции программы на языке C для процессора Core i7. Регистр EBP используется в качестве указателя кадра. Первые два слова требуются для компоновки, поэтому первый параметр `n` (или `N`, поскольку регистр символов в макроассемблере не важен) находится в ячейке EBP + 8, а за ним следуют параметры `i` и `j` в ячейках EBP + 12 и EBP + 16 соответственно. Локальная переменная `k` находится в EBP + 20.

Листинг 5.7. Решение задачи «Ханойская башня» для Core i7

```
          .686                    ; компилируется для Core i7
.MODEL FLAT
PUBLIC _towers                    ; экспорт 'towers'
EXTERN _printf:NEAR               ; импорт printf
.CODE
_towers:  PUSH EBP                ; сохраняет EBP (указатель кадра)
          MOV EBP, ESP            ; устанавливает новый
                                  ; указатель кадра над ESP
          CMP[EBP+8],1            ; if(n==1)
          JNE L1                  ; переход, если n не равно 1
          MOV EAX, [EBP+16]       ; printf("…", i, j);
          PUSH EAX                ; сохранение параметров i, j и формата
          MOV EAX, [EBP+12]       ; строка помещается в стек
          PUSH EAX                ; в обратном порядке (требование языка C)
          PUSH OFFSET FLAT:format ; OFFSET FLAT – это адрес формата
          CALL _printf            ; вызов процедуры printf
          ADD ESP, 12             ; удаление параметров из стека
          JMP Done                ; завершение
L1:       MOV EAX,6               ; начало вычисления k=6–i–j
          SUB EAX, [EBP+12]       ; EAX=6–i
          SUB EAX, [EBP+16]       ; EAX=6–i–j
          MOV [EBP+20], EAX       ; k=EAX
          PUSH EAX                ; начало процедуры towers(n–1, i, k)
          MOV EAX, [EBP+12]       ; EAX=i
          PUSH EAX                ; помещает в стек i
          MOV EAX, [EBP+8]        ; EAX=n
          DEC EAX                 ; EAX=n–1
          PUSH EAX                ; помещает в стек n–1
          CALL _towers            ; вызов процедуры towers(n–1, i, 6–i–j)
          ADD ESP, 12             ; удаляет параметры из стека
          MOV EAX, [EBP+16]       ; начало процедуры towers (1, i, j)
          PUSH EAX                ; помещает в стек j
          MOV EAX, [EBP+12]       ; EAX=i
          PUSH EAX                ; помещает в стек i
          PUSH 1                  ; помещает в стек 1
          CALL _towers            ; вызывает процедуру towers(1, i, j)
          ADD ESP, 12             ; удаляет параметры из стека
          MOV EAX, [EBP+12]       ; начало процедуры towers(n–1, 6–i–j, i)
          PUSH EAX                ; помещает в стек i
          MOV EAX, [EBP+20]       ; EAX=k
          PUSH EAX                ; помещает в стек k
          MOV EAX, [EBP+8]        ; EAX = n
          DEC EAX                 ; EAX=n–1
          PUSH EAX                ; помещает в стек n–1
          CALL _towers            ; вызов процедуры towers(n–1, 6–i–j, i)
          ADD ESP, 12             ; корректировка указателя стека
Done:     LEAVE                   ; подготовка к выходу
          RET 0                   ; возврат к вызывающей программе
.DATA
format    DB "Переместить диск с %d на %d\n" ; форматирующая строка
END
```

Процедура начинается с создания нового кадра в конце старого. Для этого значение регистра ESP копируется в указатель кадра EBP. Затем `n` сравнивается с 1,

и если n > 1, совершается переход к оператору else. Тогда код then помещает в стек три значения: адрес форматирующей строки, i и j, а затем вызывает сам себя.

Параметры помещаются в стек в обратном порядке, как того требует язык C. Указатель на форматирующую строку должен быть помещен в вершину стека. Процедура printf имеет переменное число параметров, и если параметры будут помещаться в стек в прямом порядке, то процедура не сможет узнать, в каком месте стека находится форматирующая строка.

После вызова процедуры к регистру ESP прибавляется 12, чтобы удалить параметры из стека. На самом деле они не удаляются из памяти, но изменение регистра ESP делает их недоступными через обычные операции со стеком.

Выполнение секции else начинается с метки L1. Здесь сначала вычисляется выражение 6 — i — j, а полученное значение сохраняется в переменной k. Какими бы ни были значения i и j, количество дисков на третьем колышке всегда равно 6 — i — j. Сохранение значения в переменной k избавляет от необходимости вычислять это выражение во второй раз.

Затем процедура вызывает сама себя три раза, каждый раз с новыми параметрами. После каждого вызова стек освобождается.

У начинающих программистов иногда возникают сложности. Рекурсия иногда приводит людей в замешательство. Но на самом деле она совсем не сложна. Просто параметры помещаются в стек, после чего процедура вызывает сама себя.

Решение задачи «Ханойская башня» на ассемблере OMAP4430

А теперь рассмотрим ту же программу на ассемблере OMAP4430 (листинг 5.8). Поскольку программа для OMAP4430 совершенно нечитабельна даже после длительной практики, мы решили определить несколько символических имен, чтобы прояснить дело. Чтобы такая программа работала, ее перед ассемблированием нужно пропустить через программу под названием *cpp* (препроцессор C). Здесь мы используем строчные буквы, поскольку ассемблер OMAP4430 этого требует (это на тот случай, если читатели захотят набрать эту программу и запустить).

Листинг 5.8. Решение задачи «Ханойская башня» для OMAP4430

```
#define Param0                  r0
#define Param1                  r1
#define Param2                  r2
#define FormatPtr               r0
#define                         kr7
#define n_minus_1               r5
        .text
towers:  push {r3, r4, r5, r6, r7, lr}   @ сохранение адреса возврата
         movr4, Param1                   @ и используемых регистров
         movr6, Param2
         cmp Param0, #1                  @ (n==1)?
         bne else                        @ если нет, перейти к секции else
         movw FormatPtr,#:lower16:format @ загрузить указатель
         movt FormatPtr,#:upper16:format @ на форматную строку
         bl printf                       @ print move
         pop {r3, r4, r5, r6, r7, pc}
```

```
else:   rsb k, r1, #6                   @ k=6 – i – j
        subs k, k, r2
        add n minus 1, r0, #-1          @ compute (n-1)
        movr0, n_minus_1                @ для рекурсивного вызова
        movr2, k
        bl towers                       @ call towers(n–1, i, k)
        movr0, #1
        movr1, r4
        movr2, r6
        bl towers                       @ call towers(1, k, j)
        movr0, n_minus_1
        movr1, k
        movr2, r6
        bl towers                       @ call towers(n–1, k, j)
        pop {r3, r4, r5, r6, r7, pc}    @ восстановить измененные регистры
                                        @ и вернуть управление

        .global main
main:   push {lr}                       @ сохранение адреса возврата
        movParam0, #3
        movParam1, #1
        movParam2, Param0
        bl towers                       @ call towers(3, 1, 3)
        pop {pc}                        @ извлечь адрес возврата,
                                        @ вернуть управление
format: .ascii "Move a disk from %d to %d\n\0"
```

По алгоритму версия OMAP4430 идентична версии Core i7. В обоих случаях сначала проверяется `n`, и если `n` > 1, совершается переход к `else`. Основные сложности версии OMAP4430 связаны с некоторыми особенностями архитектуры команд.

Сначала код OMAP4430 должен передать адрес форматирующей строки в `printf`, но машина не может просто переместить адрес в регистр, который содержит выходной параметр, поскольку нельзя поместить 32-разрядную константу в регистр одной командой. Для этого требуется выполнить две команды: `MOVW` и `MOVT`.

После вызова не нужно делать подстройку стека, поскольку регистровое окно автоматически корректируется командами `PUSH` и `POP` в начале и конце процедуры. Эти команды также обеспечивают сохранение и восстановление адреса возврата — при входе выполняется сохранение регистра LR, а при выходе — регистра PC.

Архитектура IA-64 и процессор Itanium 2

Около 2000 года специалисты Intel начали понимать, что скоро наступит момент, когда из линейки процессоров IA-32 будет выжато все возможное. Новые модели совершенствовались только благодаря новым технологиям производства, которые позволяли уменьшать размеры транзисторов (а значит, повышать тактовые частоты). Однако повышать скорость работы становилось все труднее, и виной тому были ограничения, свойственные архитектуре команд IA-32.

Единственное эффективное решение проблемы — при разработке новых процессоров перейти от IA-32 к новой архитектуре команд. Именно такие планы строит компания Intel. Более того, предполагается запустить две новых линейки. Первая из них, называемая EMT-64, представляет собой расширенную версию Pentium с 64-разрядными регистрами и 64-разрядным же адресным пространством. В этой архитектуре решается проблема адресного пространства, но сложности реализации, свойственные ее предшественникам, сохраняются. Ее можно назвать расширенным вариантом архитектуры Pentium.

Еще одна архитектура, которую совместно разрабатывают компании Intel и Hewlett Packard, называется **IA-64**. Это уже полноценная 64-разрядная машина, а не расширение 32-разрядной машины. Более того — она во многих отношениях разительно отличается от IA-32. Первоначально IA-64 предполагается вывести на рынок профессиональных серверных систем, но вполне возможно, что впоследствии эта архитектура укрепится и в сегменте настольных компьютеров. В любом случае, архитектура IA-64, впервые реализованная в линейке процессоров Itanium, настолько отличается от всего, что мы рассматривали ранее, что просто необходимо познакомиться с ней поближе. Итак, оставшаяся часть раздела посвящена архитектуре IA-64 как таковой и ее реализации в процессорах серии Itanium 2.

Проблема IA-32

Прежде чем переходить к детальному рассмотрению архитектуры IA-62 и процессора Itanium 2, полезно разобраться в том, чем, собственно, плоха архитектура IA-32 и какие проблемы компания Intel намеревается решить путем разработки новой архитектуры. Основная проблема заключается в том, что IA-32 — это старая архитектура команд с совершенно неподходящими для современных технологий свойствами. Это типичная CISC-архитектура с командами разной длины и огромным количеством различных форматов, которые трудно декодировать быстро и «на лету». Современная техника лучше всего работает с RISC-архитектурами, в которой команды одинаковы по размеру, а код операции имеет фиксированную длину, поэтому его легко декодировать. Хотя команды архитектуры IA-32 во время выполнения программы можно разделить на микрооперации наподобие RISC-команд, но для этого требуется дополнительное аппаратное обеспечение (пространство на микросхеме), к тому же это занимает время и усложняет разработку. Это первый недостаток.

IA-32 — это архитектура, ориентированная на двухадресные команды и работу с памятью. В настоящее время популярны архитектуры команд типа загрузки/сохранения, в которых обращения к памяти выполняются только для того, чтобы поместить операнды в регистры, а все вычисления делаются с использованием трехадресных регистровых команд. Поскольку скорость работы процессора растет гораздо быстрее, чем памяти, положение дел с IA-32 со временем все больше ухудшается. Это второй недостаток.

Архитектура IA-32 содержит небольшой и нерегулярный набор регистров. Это затрудняет разработку компиляторов, к тому же из-за малого числа регистров общего назначения (четыре или шесть, в зависимости от того, куда отнести регистры ESI и EDI) постоянно приходится записывать в память промежуточные результаты, что ведет к дополнительным обращениям к памяти, даже когда они по логике вещей не нужны. Это третий недостаток.

Из-за недостаточного числа регистров возникает множество ситуаций зависимостей, особенно WAR-зависимостей, поскольку промежуточные результаты нужно куда-то помещать, а дополнительных регистров нет. Из-за недостатка регистров постоянно требуется их подмена (то есть использование скрытых регистров). Во избежание слишком частых кэш-промахов команды приходится выполнять не по порядку. Однако семантика архитектуры IA-32 определяет точные прерывания, поэтому команды, выполняемые не по порядку, должны записывать результаты в выходные регистры в строгом порядке. Все это очень сложно реализовать аппаратно. Это четвертый недостаток.

Чтобы скорость работы была высокой, система должна быть в значительной степени конвейеризированна. Однако это значит, что для выполнения любой команды требуется множество циклов. Следовательно, становится существенным точное предсказание переходов, поскольку в конвейер должны попадать только нужные команды. Однако даже если процент неправильных прогнозов невысок, существенно снижается производительность. Это пятый недостаток.

Чтобы избежать проблем с неправильным прогнозированием переходов, процессору приходится осуществлять спекулятивное выполнение команд со всеми вытекающими отсюда последствиями. Это шестой недостаток.

Мы не будем перечислять недостатки дальше, поскольку и так ясно, что за ними кроется реальная проблема. А ведь мы еще не упомянули, что 32-разрядные адреса архитектуры IA-32 ограничивают размер отдельных программ значением 4 Гбайт, что является серьезной проблемой для высокопроизводительных серверов. Допустим, эта проблема решается в архитектуре EMT-64, но остальные недостатки остаются.

Ситуацию с IA-32 можно сравнить с положением дел в небесной механике как раз перед появлением Коперника. В те времена в астрономии доминировала геоцентрическая теория, в соответствие с которой Земля является центром вселенной и неподвижна, а планеты движутся вокруг нее. Однако новые наблюдения показывали все больше и больше несоответствий этой теории действительности, пока в конце концов старая модель не рухнула под весом собственной внутренней сложности.

Компания Intel находится приблизительно в таком же положении. Множество транзисторов в процессоре Core i7 предназначено исключительно для переделки CISC-команд в RISC-команды, разрешения конфликтов, прогнозирования переходов, исправления последствий неправильных прогнозов и решения многих других задач подобного рода, а для реальной работы, которая собственно и нужна пользователю, остается лишь незначительная часть этих транзисторов . Поэтому компания Intel пришла к следующему выводу: нужно выбросить IA-32 на помойку и начать все заново (IA-64). Архитектура EMT-64 призвана лишь выиграть некоторое время, оставляя проблему нерешенной.

Модель IA-64 — вычисления с явным параллелизмом команд

Основной принцип организации архитектуры IA-64 сводится к тому, чтобы перенести нагрузку с периода исполнения в период компиляции. Процессор Core i7 в ходе исполнения переупорядочивает команды, подменяет регистры, распреде-

ляет функциональные блоки и выполняет множество других функций, что ведет к максимальной загрузке всех аппаратных ресурсов. В модели IA-64 эти задачи заранее решает компилятор. В результате он генерирует программу, которую можно исполнять без излишних манипуляций аппаратными средствами. К примеру, в Core i7 компилятор действует так, словно в машине всего 8 регистров, хотя на самом деле их 128, в результате во время выполнения программы приходится как-то выкручиваться, чтобы избежать взаимозависимостей. Согласно архитектуре IA-64, компилятор получает достоверную информацию о количестве регистров в машине, а затем генерирует программу, в которой нет никаких конфликтов между регистрами. Кроме того, компилятор следит за загрузкой функциональных блоков и не запускает команды, в которых предполагается обращение к занятым функциональным блокам. Модель, в которой аппаратный параллелизм является видимым для компилятора, называется **EPIC** (Explicitly Parallel Instruction Computing — **вычисления с явным параллелизмом команд**). В определенной степени модель EPIC можно считать развитием RISC-технологии.

Некоторые особенности IA-64 заметно повышают производительность. Среди них — сокращение числа обращений к памяти, планирование команд, сокращение числа условных переходов и спекулятивные операции. Все эти особенности мы обсудим как с теоретической точки зрения, так и в контексте их реализации в Itanium 2.

Сокращение числа обращений к памяти

Модель памяти Itanium 2 довольно проста. Всего предусмотрено 2^{64} байт линейной памяти. Имеющиеся команды позволяют обращаться к блокам памяти размером 1, 2, 4, 8, 16 и 10 байт (последнее значение введено для совместимости с 80-разрядными числами с плавающей точкой стандарта IEEE 745). Категорической необходимости в выравнивании обращений к памяти по естественным границам нет, однако без выравнивания производительность ниже. Память может быть как с прямым, так и с обратным порядком следования байтов; тот или иной формат устанавливается специальным битом в регистре, загружаемом операционной системой.

Работа с памятью в современных компьютерах считается узким местом. Связано это с тем, что процессоры работают гораздо быстрее модулей памяти. Сократить число обращений к памяти можно путем размещения большого кэша первого уровня на микросхеме процессора и еще большего кэша второго уровня в непосредственной близости от микросхемы. Двумя модулями кэш-памяти оснащаются все современные процессоры. В то же время существуют и другие методы, позволяющие сократить объем взаимодействия с памятью, и некоторые из них реализованы в IA-64.

Лучший способ ускорить обращения к памяти — выполнять эту операцию в фоновом режиме. В процессоре Itanium 2 предусмотрено 128 64-разрядных регистров общего назначения. Первые 32 из них являются статическими, а оставшиеся 96 группируются в стек регистров, напоминающий регистровое окно других RISC-процессоров (например, UltraSPARC). В отличие от UltraSPARC количество доступных программе регистров меняется от одной процедуры к другой. В итоге каждая процедура получает доступ к 32 статическим регистрам и некоторому (переменному) количеству регистров, распределяемых динамически.

При вызове процедуры указатель стека регистров смещается таким образом, чтобы входные параметры оказались видимыми в регистрах, но сами регистры не были распределены между локальными переменными. Процедура сама определяет количество необходимых ей регистров и соответствующим образом перемещает указатель стека. Сохранять содержимое этих регистров при входе и восстанавливать при выходе не требуется, хотя, если процедуре нужно изменить статический регистр, она должна сначала явно сохранить его прежнее значение, а впоследствии восстановить его. Поскольку количество регистров выражено доступной переменной и обуславливается требованиями каждой конкретной процедуры, неэффективное применение регистров исключается. Кроме того, увеличивается максимальная глубина вызова процедур, при которой регистры не требуется «сбрасывать» в память.

В Itanium 2 есть 128 регистров с плавающей точкой, организованных по стандарту IEEE 745 и не группируемых в стек. Большое количество регистров этого типа позволяет сохранять промежуточные результаты множества операций с плавающей точкой в регистрах, не перемещая их в память.

Кроме того, в Itanium 2 предусмотрено 64 1-разрядных предикатных регистра, 8 регистров переходов и 128 специализированных прикладных регистров, которые используются для самых различных целей, в частности для обмена параметрами между прикладными программами и операционной системой. Общая схема регистров Itanium 2 представлена на рис. 5.28.

Рис. 5.28. Регистры Itanium 2

Планирование команд

Один из наиболее серьезных недостатков Core i7 — сложность планирования команд для обработки в различных функциональных блоках без взаимозависимостей. Для решения этой проблемы в период исполнения задействуются очень сложные механизмы, аппаратная поддержка которых требует много места на микросхеме. В архитектуре IA-64 и ее реализации — Itanium 2 — эти проблемы решены путем передачи соответствующих функций компилятору. Каждая программа теперь состоит из последовательности **групп команд**. Команды в рамках одной группы не конфликтуют друг с другом, не потребляют больше ресурсов и не обращаются к большему количеству функциональных блоков,

чем предусмотрено системой, не образуют RAW- и WAW-взаимозависимости (WAR-взаимозависимости допускаются в ограниченном объеме). Создается впечатление, что последовательные группы команд исполняются строго по порядку, хотя на самом деле, если это безопасно, процессор может запустить часть команд следующей группы до завершения предыдущей.

Таким образом, процессор может планировать исполнение команд внутри каждой отдельно взятой группы в любом порядке, по возможности, — в параллельном режиме. При этом вероятность возникновения конфликтов между командами равна нулю. Если группа команд нарушает вышеуказанные правила, поведение программы становится неопределенным. В такой ситуации обязанность по обеспечению соответствия полученного из исходной программы кода на ассемблере всем требованиям ложится на компилятор. Для ускорения компиляции в процессе отладки программы компилятор может поместить каждую команду в отдельную группу; сделать это просто, но производительность в результате этой операции снижается. В процессе генерировании окончательной версии кода компилятор тратит значительное время на его оптимизацию.

Команды объединяются в 128-разрядные **пучки** (bundles), подобные изображенному в верхней части рис. 5.29. В каждом пучке содержится три 41-разрядных команды и 5-разрядный шаблон. Количество пучков в группе команд не всегда выражается целым числом — в некоторых случаях группы начинаются и заканчиваются посередине пучков.

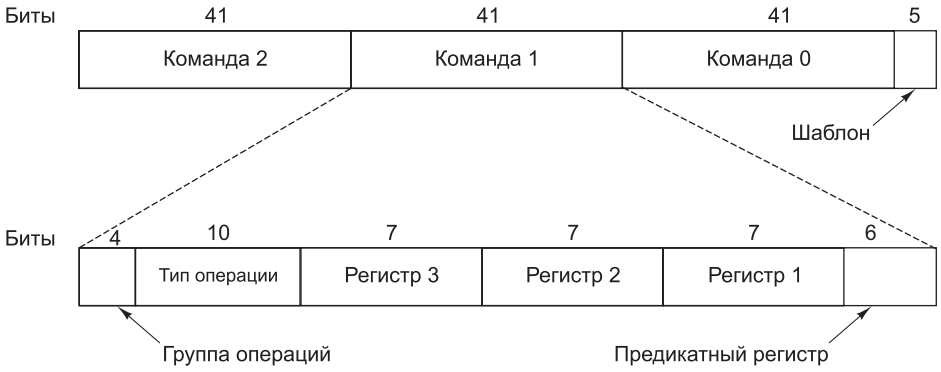


Рис. 5.29. Пучок в архитектуре IA-64 состоит из трех команд

Существуют более 100 форматов команд. На рис. 5.29 представлен наиболее распространенный формат. В данном случае он применяется для выполнения при помощи АЛУ операции `ADD`, которая размещает сумму двух регистров в третьем. Поле группы операций определяет общий класс команды (например, целочисленная операция АЛУ). В поле типа операции указывается конкретная операция (например, `ADD` или `SUB`). Далее следуют три поля регистра. Последнее поле данного формата — поле предикатного регистра — мы обсудим чуть позже.

Шаблон пучка указывает, какие функциональные блоки нужны для его обработки, а также определяет положение границы группы команд (если таковая предусмотрена). Основными функциональными блоками считаются целочисленные и нецелочисленные команды АЛУ, обращения к памяти, операции с плавающей точкой, переходы и т. д. Очевидно, что при наличии шести блоков

и трех команд для полной ортогональности требуются 216 базовых комбинаций плюс 3 × 216 дополнительных комбинаций для определения маркеров группы после команд 0, 1 и 2. Поскольку доступно всего 5 бит, из всего разнообразия комбинаций допускаются лишь некоторые. С другой стороны, если бы в состав пучка можно было включить сразу три команды с плавающей точкой, процессор просто не смог бы их одновременно запустить. Таким образом, допустимыми считаются только фактически осуществимые комбинации.

Сокращение числа условных переходов — предикация

Еще одна особенность архитектуры IA-64 — новый способ обработки условных переходов. Если бы была возможность избавиться от большинства из них, центральный процессор стал бы гораздо проще и работал бы гораздо быстрее. На первый взгляд может показаться, что устранить условные переходы невозможно, поскольку в программах всегда полно операторов `if`. Однако в архитектуре IA-64 используется специальная технология, названная **предикацией** (predication), которая позволяет сильно сократить их число [August et al., 1998; Hwu, 1998]. Кратко опишем эту технологию.

В нынешних машинах все команды являются безусловными в том смысле, что когда центральный процессор встречает команду, он просто ее выполняет. Здесь никогда не решается вопрос: «Выполнять или не выполнять?» И напротив, в предикатной архитектуре команды содержат условия, которые сообщают, в каком случае нужно выполнять команду, а в каком — нет. Именно этот переход от безусловных команд к предикатным позволяет избавиться от многих условных переходов. Вместо того чтобы выбирать ту или иную последовательность безусловных команд, все команды сливаются в одну последовательность предикатных команд, в которой у разных команд разные предикаты.

Чтобы понять, как работает предикация, рассмотрим простой пример (листинги 5.9–5.11), в котором показано **условное выполнение** команд (условное выполнение — предтеча предикации). В листинге 5.9 мы видим оператор `if`. В листинге 5.10 после его трансляции получилось три команды: сравнения, условного перехода и перемещения. В листинге 5.11 мы избавились от условного перехода, используя новую команду `CMOVZ`, которая является командой условного перемещения. Эта команда проверяет, равен ли третий регистр R1 нулю. Если равен, то команда копирует R3 в R2, а если нет — команда не выполняет никаких действий.

Листинг 5.9. Оператор if

```
if (R1==0)
        R2 = R3;
```

Листинг 5.10. Код на ассемблере для листинга 5.9

```
CMP R1,0
BNE L1
MOV R2, R3
L1:
```

Листинг 5.11. Условная команда

```
CMOVZ R2,R3,R1
```

Если у нас есть команда, которая может копировать данные, когда какой-либо регистр равен нулю, значит, у нас может быть и такая команда, которая копирует данные, если какой-нибудь регистр не равен нулю. Пусть это будет команда CMOVN. При наличии обеих команд мы уже на пути к полному условному выполнению. Представим оператор if с несколькими операторами присваивания в части then и несколькими операторами присваивания в части else. Весь этот фрагмент программы можно транслировать в код, который будет устанавливать какой-нибудь регистр на 0, если условие не выполнено, и на какое-нибудь другое значение, если условие выполнено. Таким образом, присваивания в части then можно скомпилировать в последовательность команд CMOVN, а присваивания в части else — в последовательность команд CMOVZ.

Все эти команды, в том числе команды установки регистров, CNOVN и CMOVZ, формируют единый основной блок без условных переходов. Команды можно даже переупорядочить при компиляции или во время выполнения программы. Единственное требование состоит в том, чтобы условие было известно к тому моменту, когда условные команды потребуется помещать в выходные регистры (то есть где-то в конце конвейера). Простой пример фрагмента программы с операторами then и else приведен в листингах 5.12–5.14.

Листинг 5.12. Оператор if

```
if(R1 == 0) {
  R2 = R3;
  R4 = R5;
} else {
  R6 = R7;
  R8 = R9;
}
```

Листинг 5.13. Код на ассемблере для листинга 5.12

```
        CMP R1,0
        BNE L1
        MOV R2,R3
        MOV R4,R5
        BR L2
L1:     MOV R6,R7
        MOV R8,R9
L2:
```

Листинг 5.14. Условное выполнение

```
CMOVZ R2,R3,R1
CMOVZ R4,R5,R1
CMOVN R6,R7,R1
CMOVN R8,R9,R1
```

Мы показали только очень простые условные команды (взятые из архитектуры команд IA-32), но в архитектуре IA-64 все команды предикатные. Это значит, что выполнение каждой команды можно сделать условным. Дополнительное 6-разрядное поле предикатного регистра, о котором мы упомянули, позволяет выбрать один из 64 1-разрядных предикатных регистров. Следовательно, оператор if может быть скомпилирован в код, который устанавливает один из предикатных регистров в 1, если условие истинно, и в 0, если условие ложно. Одновременно и автоматически инвертируется другой предикатный регистр. Таким образом, при

поддержке предикации машинные команды, которые формируются из операторов `then` и `else`, сливаются в единый поток команд, причем у команд первого из них поле предикатного регистра оказывается единичным, у второго — нулевым. При передаче управления будет выполнен только один набор команд.

В листингах 5.15–5.17 показано, как предикация используется для устранения переходов. Команда `CMPEQ` сравнивает два регистра и устанавливает предикатный регистр P4 в 1, если они равны, и в 0, если они не равны. Кроме того, команда инвертирует еще один регистр, например P5. Этого команды частей `if` и `then` можно поместить одну за другой, причем каждая из них оказывается связанной с каким-нибудь предикатным регистром (регистр указывается в угловых скобках). Сюда можно поместить любой код, при условии, что каждая команда предсказывается правильно.

Листинг 5.15. Оператор if

```
if(R1 == R2)
        R3 = R4 + R5;
Else
        R6 = R4 – R5
```

Листинг 5.16. Код на ассемблере для листинга 5.15

```
        CMP R1,R2
        BNE L1
        MOV R3,R4
        ADD R3,R5
        BR L2
L1:     MOV R6,R4
        SUB R6,R5
L2:
```

Листинг 5.17. Предикатное выполнение

```
CMPEQ R1,R2,P4
<P4> ADD R3,R4,R5
<P5> SUB R6,R4,R5
```

В архитектуре IA-64 эта идея доведена до логического завершения — здесь с предикатными регистрами связаны и команды сравнения, и арифметические команды, и некоторые другие команды. Предикатные команды могут помещаться в конвейер последовательно без каких-либо проблем и простоев. Поэтому они очень полезны.

В архитектуре IA-64 предикация происходит следующим образом. Каждая команда действительно выполняется, и в самом конце конвейера, когда уже нужно сохранять результат в выходном регистре, производится проверка, истинно ли предсказание. Если да, то результаты просто записываются в выходной регистр. Если предсказание ложно, то записи в выходной регистр не происходит. Подробно о предикации вы можете прочитать в дополнительной литературе [Dulong, 1998].

Спекулятивная загрузка

Еще одна особенность IA-64, повышающая быстродействие, — поддержка спекулятивной загрузки. Если спекулятивная команда `LOAD` не срабатывает, то вместо того чтобы вызвать исключение, она просто прекращает выполняться и сообщает, что регистр, в который она должна была загрузить значение, недействителен. Для

этого используется тот самый бит отравления, о котором мы упоминали в главе 4. А исключение будет вызвано только в том случае, если затем попытаться использовать этот регистр.

Обычно при спекулятивной загрузке компилятор помещает команды `LOAD` перед другими командами. Поскольку выполнение этих команд начинается раньше, чем нужно, они могут завершиться еще до того, как потребуются результаты. В том месте, где ему нужно получить значение определенного регистра, компилятор вставляет команду `CHECK`. Если значение там уже есть, команда `CHECK` работает так же, как `NOP`, и выполнение программы просто сразу продолжается дальше. Если значения в регистре еще нет, следующая команда вынуждена простаивать.

Суммируя, можно сказать, что в машинах с архитектурой IA-64 реализовано несколько механизмов повышения быстродействия. Во-первых, это современная конвейеризированная трехадресная RISC-машина, поддерживающая механизм загрузки/сохранения. Уже один этот фактор является значительным усовершенствованием по сравнению с излишней сложностью архитектуры IA-32.

Во-вторых, IA-64 поддерживает модель явного параллелизма. Компилятор определяет, какие команды могут выполняться одновременно, и не вступая в конфликт, группирует эти команды в пучки. Таким образом, процессор может просто планировать обработку пучков, не думая ни о каких проверках. Перемещение работы со стадии исполнения на стадию компиляции всегда эффективно.

В-третьих, предикация позволяет объединить команды обоих переходов в операторе `if`, устраняя при этом как условный переход, так и необходимость прогнозирования этого перехода. Наконец, спекулятивная загрузка позволяет вызывать операнды заранее, и даже если позднее окажется, что эти операнды не нужны, ничего страшного не произойдет.

В общем и целом архитектура Itanium хорошо продумана, и она соответствует интересам проектировщиков и пользователей. Итак, процессор Itanium работает в вашем компьютере или в компьютере вашего соседа? Ответ: нет, нет и еще раз (скорее всего) нет. Сегодня, спустя уже более 10 лет с момента выхода процессора Itanium, его популярность можно описать в лучшем случае как весьма умеренную. Но компания Intel продолжает производить системы на базе Itanium, пусть и ограниченные высокопроизводительными серверами.

Итак, вернемся к исходным проблемам, обусловившим создание архитектуры IA-64. Процессор Itanium создавался для исправления недостатков архитектуры IA-32. Так как широкого распространения он не получил, как же Intel решает эти проблемы? Как будет показано в главе 8, развитие IA-32 основано не на переработке архитектуры набора команд, а на активной реализации параллельных вычислений на базе многопроцессорных архитектур. Дополнительные сведения о процессоре Itanium 2 и его микроархитектуре можно почерпнуть в дополнительной литературе [McNairy and Soltis, 2003; Rusu et al., 2004].

Краткое содержание главы

Для большинства людей уровень архитектуры набора команд — это «машинный язык», хотя на CISC-компьютерах он обычно строится поверх более низкого

уровня микрокода. На этом уровне машина имеет память с байтовой или пословной организацией, состоящую из нескольких десятков мегабайтов и содержащую команды наподобие `MOVE`, `ADD` и `BEQ`.

В большинстве современных компьютеров память организована в виде последовательности байтов, при этом 4 или 8 байт группируются в слова. Обычно в машине есть от 8 до 32 регистров, каждый из которых содержит одно слово. В некоторых машинах (например, в Core i7) при обращении к словам памяти выравнивание по естественным границам ячеек не требуется, в других (например, в OMAP4430) это — обязательное условие. Впрочем, даже если выравнивание не является обязательным условием, с ним операции выполняются быстрее.

Команды обычно имеют 1, 2 или 3 операнда, обращение к которым происходит с помощью различных режимов адресации: непосредственной, прямой, регистровой, индексной и т. д. Некоторые машины поддерживают обширный набор сложных режимов адресации. Достаточно часто компиляторы не могут эффективно использовать эти режимы. Команды обычно могут перемещать данные, выполнять унарные и бинарные операции (в том числе арифметические и логические), совершать переходы, вызывать процедуры, выполнять циклы, а иногда и некоторые операции ввода-вывода. Типичные команды перемещают слово из памяти в регистр или наоборот, складывают, вычитают, умножают или делят два регистра или регистр и слово из памяти или сравнивают два значения в регистрах или памяти. Довольно часто количество команд в компьютерах превышает 200. В CISC-процессорах их и того больше.

Для передачи управления на уровне архитектуры команд используются различные примитивы: перехода, вызовов процедур и сопрограмм, перехвата исключений и обработки прерываний. Переходы нужны для того, чтобы остановить одну последовательность команд и начать новую (возможно, находящуюся в памяти на значительном расстоянии от первой). Процедуры позволяют выделить какой-то фрагмент программы, который можно затем вызывать из различных мест этой же программы. Сопрограммы позволяют двум потокам управления работать параллельно. Перехват исключений используется для сигнализации об исключительных ситуациях (например, о переполнении). Механизм прерываний дает возможность выполнять ввод-вывод параллельно с основными вычислениями, при этом, как только ввод-вывод завершается, центральный процессор получает сигнал об этом.

Задачу «Ханойская башня» можно решить с использованием рекурсии. Также существуют решения, основанные на итерации, но они значительно сложнее и менее элегантны, чем рассмотренное нами рекурсивное решение.

Наконец, в архитектуре IA-64 используется вычислительная модель EPIC, упрощающая реализацию параллелизма в программах. Для повышения быстродействия в этой архитектуре предусмотрены группировка команд, предикация и спекулятивная загрузка. Архитектура IA-64 способна стать удачной заменой Core i7, даже несмотря на то, что она возлагает на компилятор большую нагрузку в плане поддержания параллелизма. Тем не менее выполнение работы на стадии компилирования всегда предпочтительнее ее выполнения на стадии исполнения.

Вопросы и задания

1. Слову в системе с прямым порядком следования байтов присвоено численное значение 3. Предположим, что это слово байт за байтом передается в систему с обратным порядком следования байтов и сохраняется в ней, причем исходный байт 0 соответствует целевому байту 0 и т. д. Каким станет численное значение слова в системе с обратным порядком следования байтов?
2. Прежде во многих операционных системах и компьютерах использовались раздельные пространства команд и данных, благодаря чему k-разрядный адрес мог обозначать до 2^k программных адресов и 2^k адресов данных. Например, для $k = 32$ программа могла обращаться к 4 Гбайт команд и 4 Гбайт данных; общий объем адресного пространства достигал 8 Гбайт. Так как при использовании этой схемы программа не может модифицировать себя в памяти, как операционная система могла организовать загрузку программ в память?
3. Разработайте расширенный код операций, который позволяет закодировать в 32-разрядной команде следующее:
   - 15 команд с двумя 12-разрядными адресами и 4-разрядным номером регистра;
   - 650 команд с одним 12-разрядным адресом и 4-разрядным номером регистра;
   - 80 команд без адресов и регистров.
4. Пусть в машине поддерживаются 16-разрядные команды и 6-разрядные адреса. Одни команды содержат один адрес, другие — два. Если существуют n двухадресных команд, то каково максимальное число одноадресных команд?
5. Можно ли разработать такой расширенный код операций, который позволял бы кодировать в 12-разрядной команде следующее (регистр представляется 3 битами):
   - 4 команды с тремя регистрами;
   - 255 команд с одним регистром;
   - 16 команд без регистров.
6. Пусть имеется одноадресная машина с регистром-сумматором. Вот значения некоторых слов памяти:
   - слово 20 содержит число 40;
   - слово 30 содержит число 50;
   - слово 40 содержит число 60;
   - слово 50 содержит число 70;

   Какие значения загрузят в регистр-сумматор перечисленные ниже команды?

```
LOAD IMMEDIATE 20
LOAD DIRECT 20
LOAD INDIRECT 20
LOAD IMMEDIATE 30
LOAD DIRECT 30
LOAD INDIRECT 30
```

7. Для каждого из четырех видов машин — безадресной, одноадресной, двухадресной и трехадресной — напишите программу вычисления следующего выражения:

   $X = (A + B \times C)/(D - E \times F)$

   В наличии имеются следующие команды.

   - безадресные: `PUSH` M, `POP` M, `ADD`, `SUB`, `MUL`, `DIV`;
   - одноадресные: `LOAD` M, `STORE` M, `ADD` M, `SUB` M, `MUL` M, `DIV` M;
   - двухадресные: `MOV` ($X = Y$), `ADD` ($X = X + Y$), `SUB` ($X = X - Y$), `MUL` ($X = X \times Y$), `DIV` ($X = X/Y$);
   - трехадресные: `MOV` ($X = Y$), `ADD` ($X = Y + Z$), `SUB` ($X = Y - Z$), `MUL` ($X = Y \times Z$), `DIV` ($X = Y/Z$).

   Здесь M — это 16-разрядный адрес памяти, а X, Y и Z — либо 16-разрядные адреса, либо 4-разрядные регистры. Безадресная машина использует стек, одноадресная машина — регистр-аккумулятор, а оставшиеся две имеют 16 регистров и команды, которые оперируют всеми комбинациями ячеек памяти и регистров. Команда `SUB X,Y` вычитает Y из X, а команда `SUB X,Y,Z` вычитает Z из Y и помещает результат в X. Если длина кодов операций равна 8 бит, а размеры команд кратны величине 4 бит, сколько битов нужно каждой машине для вычисления X?

8. Придумайте такой механизм адресации, который позволяет определять в 6-разрядном поле произвольный набор из 64 адресов, не обязательно смежных.
9. В чем недостаток самомодифицирующихся программ, о котором не было упомянуто в тексте этой главы?
10. Переделайте следующие формулы из инфиксной записи в обратную польскую запись:

    1) $A + B + C + D$ - E;
    2) $(A + B) \times (C + D) + E$;
    3) $(A \times B)+(C \times D) + E$;
    4) $(A - B) \times (((C - D \times E)/F)/G) \times H$.

11. Какие из следующих пар формул в обратной польской записи математически эквивалентны?

    1) $A\ B + C +$ и $A\ B\ C + +$;
    2) $A\ B - C -$ и $A\ B\ C - -$;
    3) $A\ B \times C+$ и $A\ B\ C + \times$.

12. Переделайте следующие формулы из обратной польской записи в инфиксную запись:

    1) $A\ B + C + D \times$;
    2) $A\ B\ /\ C\ D\ /\ +$;
    3) $A\ B\ C\ D\ E + \times \times\ /$;
    4) $A\ B\ C\ D\ E \times F\ /\ + G - H\ / \times +$.

13. Напишите три формулы в обратной польской записи, которые нельзя переделать в инфиксную запись.
14. Переделайте следующие инфиксные логические формулы в обратную польскую запись:
    1) (*A* И *B*) ИЛИ *C*;
    2) (*A* ИЛИ *B*) И (*A* ИЛИ *C*);
    3) (*A* И *B*) ИЛИ (*C* И *D*).
15. Переделайте следующую инфиксную формулу в обратную польскую запись и напишите IJVM-код, чтобы ее выполнить:
    $(5 \times 2 + 7) - (4/2 + 1)$.
16. Сколько регистров в машине, форматы команд которой представлены на рис. 5.17?
17. В форматах команд на рис. 5.17 варианты 1 и 2 формата различаются по биту 23. Однако для определения варианта 3 формата никакого специального бита не предусмотрено. Как аппаратное обеспечение узнает, что нужен вариант 3?
18. В программировании часто встречается задача проверки принадлежности переменной `X` интервалу от `A` до `B`. Если бы имелась трехадресная команда с операндами `A`, `B` и `X`, сколько битов кода условия было бы установлено этой командой?
19. Опишите одно преимущество и один недостаток механизма адресации относительно счетчика команд.
20. Corei7 содержит бит кода условий, состояние которого зависит от переноса бита 3 после выполнения арифметической операции. Зачем это нужно?
21. Один из ваших друзей стучится к вам в комнату в 3 часа ночи и радостно сообщает, что у него появилась замечательная идея — создать команду с двумя кодами операций. Что вы сделаете в этой ситуации, отправите своего друга получать патент или пошлете его (думать дальше)?
22. В программировании очень распространены следующие формы проверки:
    ```
    if (k==0) ...
    if (a>b) ...
    if (k<5) ...
    ```
    Предложите команду, которая будет эффективно проверять эти условия. Какие поля будут в вашей команде?
23. Покажите, каким будет 16-разрядное двоичное число 1001 0101 1100 0011 после:
    1) сдвига вправо на 4 бита с заполнением нулями;
    2) сдвига вправо на 4 бита с расширением по знаку;
    3) сдвига влево на 4 бита;
    4) циклического сдвига влево на 4 бита;
    5) циклического сдвига вправо на 4 бита.
24. Как в машине, в которой нет команды `CLR`, очистить слово памяти?
25. Вычислите логическое выражение (*A* И *B*) ИЛИ *C* для:
    - *A* = 1101 0000 1010 1101;

- *B* = 1111 1111 0000 1111;
- *C* = 0000 0000 0010 0000.

26. Придумайте, как поменять местами две переменные `A` и `B`, не используя при этом третьей переменной или регистра. Подсказка: подумайте о команде ИСКЛЮЧАЮЩЕЕ ИЛИ.
27. На некоем компьютере можно перемещать число из одного регистра в другой, сдвигать каждый из них влево на разное количество байтов и складывать полученные результаты быстрее, чем при умножении. При каком условии эта последовательность команд будет полезна для вычисления произведения «константа × переменная»?
28. Разные машины имеют разную плотность команд (то есть разное число байтов, которое требуется для выполнения определенного вычисления). Транслируйте следующие три фрагмента Java-программы на ассемблер Core i7 и IJVM. Затем посчитайте, сколько байтов требуется для выполнения каждого выражения для каждой машины. Предполагается, что `i` и `j` — это локальные переменные в памяти. В остальном отталкивайтесь от наиболее оптимистичных допущений.

```
i =3;
i = j;
i = j – 1;
```

29. В этой главе рассматривались команды для циклов `for`. Разработайте команду для циклов `while`.
30. Предположим, что ханойские монахи могут перемещать один диск за одну минуту (они не торопятся закончить работу, поскольку в Ханое очень мало вакансий для людей с подобными навыками). Сколько времени им потребуется, чтобы решить задачу (то есть переместить все 64 диска)? Ответ дайте в годах.
31. Почему устройства ввода-вывода помещают вектор прерываний на шину? Разве нельзя вместо этого сохранить соответствующую информацию в таблице в памяти?
32. Компьютер для считывания информации с диска использует канал прямого доступа к памяти. Диск содержит 64 сектора по 512 байт на дорожке. Время оборота диска составляет 16 мс. Ширина шины — 16 бит. Каждая передача шины занимает 500 нс. В среднем для одной команды процессора требуется два цикла шины. Насколько скорость работы процессора замедляется из-за прямого доступа к памяти?
33. Передача данных DMA, представленная на рис. 5.20, требует двух пересылок данных между устройством ввода-вывода и памятью. Опишите, как можно улучшить производительность DMA за счет использования архитектуры шины, показанной на рис. 3.33.
34. Почему программам обработки прерываний приписываются приоритеты, а обычные процедуры приоритетов не имеют?
35. Архитектура IA-64 предусматривает необычайно большое число регистров (64). Связано ли это с предикацией? Если да, то каким образом? Если нет, то зачем тогда их так много?

36. В этой главе обсуждалось понятие спекулятивной загрузки. Но о командах спекулятивного сохранения мы не упоминали. Почему? Может быть, они просто аналогичны командам спекулятивной загрузки? Или существует какая-то другая причина, по которой мы не стали о них говорить?
37. Когда нужно связать две локальные сети, между ними помещается мост, соединенный с обеими сетями. Каждый передаваемый какой-либо сетью пакет вызывает прерывание на мосту, чтобы мост мог определить, нужно ли этот пакет пересылать. Предположим, что на обработку прерывания и проверку пакета требуется 250 мкс, но пересылка этого пакета в случае необходимости совершается путем прямого доступа к памяти, поэтому центральный процессор от этой работы освобожден. Если размер всех пакетов равен 1 Кбайт, какова максимальная скорость передачи данных на каждой из сетей без потери пакетов?
38. На рис. 5.25 указатель кадра указывает на первую локальную переменную. Какая информация нужна программе для выхода из процедуры и возвращения к исходному состоянию?
39. Напишите подпрограмму на ассемблере для преобразования целого двоичного числа со знаком в ASCII-код.
40. Напишите подпрограмму на ассемблере для преобразования инфиксной формулы в обратную польскую запись.
41. Процедура для решения задачи «Ханойская башня» — не единственная рекурсивная процедура, любимая многими компьютерщиками. Есть еще одна очень популярная рекурсивная процедура $n!$, где $n! = n(n - 1)!$, которая подчиняется ограничивающему условию $0! = 1$. Напишите на вашем любимом ассемблере процедуру для вычисления $n!$.
42. Если вы еще не убедились в полезности рекурсии, попробуйте написать программу для решения задачи «Ханойская башня» без рекурсии и без ее имитации путем хранения стека в массиве. (Имейте в виду, что решения вы, по всей вероятности, найти не сможете.)

Глава 6
Уровень операционной системы

Как уже отмечалось, современный компьютер организован в виде иерархии уровней, каждый из которых добавляет определенные функции к нижележащему уровню. Мы рассмотрели цифровой логический уровень, уровень микроархитектуры и уровень архитектуры команд. Настало время перейти к следующему уровню — уровню операционной системы.

С точки зрения программиста, **операционная система** — это программа, добавляющая ряд команд и функций к командам и функциям, предлагаемым уровнем архитектуры команд. Обычно операционная система реализуется программно, но нет никаких веских причин, по которым ее, как микропрограммы, нельзя было бы реализовать аппаратно. Уровень операционной системы показан на рис. 6.1.

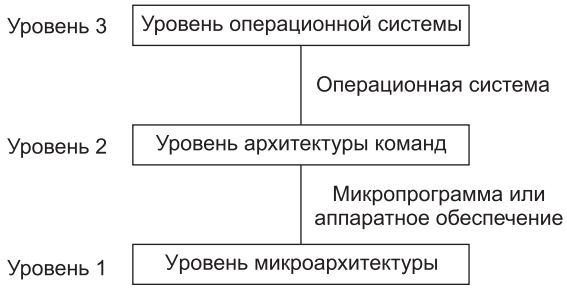


Рис. 6.1. Положение уровня операционной системы в иерархии

Хотя и уровень операционной системы, и уровень архитектуры команд абстрактны (в том смысле, что не являются реальными устройствами), между ними есть важное различие. Все команды уровня операционной системы доступны для прикладных программистов. Это — практически все команды более низкого уровня, а также новые команды, добавленные операционной системой. Новые команды называются **системными вызовами**. Они вызывают предопределенную службу операционной системы, в частности одну из ее команд. Например, типичный системный вызов может читать данные из файла.

Уровень операционной системы всегда интерпретируется. Когда пользовательская программа вызывает команду операционной системы, например чтение данных из файла, операционная система выполняет эту команду шаг за шагом, точно так же, как микропрограмма выполняет команду `ADD`. Однако когда программа вызывает команду уровня архитектуры команд, эта команда выполняется непосредственно уровнем микроархитектуры без участия операционной системы.

В этой книге мы можем рассказать об уровне операционной системы лишь в общих чертах. Мы сосредоточимся на трех важных особенностях. Первая особенность — это виртуальная память. Механизм виртуальной памяти используется многими операционными системами. Она позволяет создать впечатление, будто

у машины больше памяти, чем есть на самом деле. Вторая особенность — файловый ввод-вывод. Это понятие более высокого уровня, чем команды ввода-вывода, которые мы рассматривали в предыдущей главе. Третья особенность — параллелизм (как организовано одновременное выполнение нескольких процессов, обмен информацией и синхронизация). Понятие процесса является очень важным, и мы подробно рассмотрим его далее в этой главе. Под процессом можно понимать работающую программу и всю информацию об ее состоянии (памяти, регистрах, счетчике команд, вводе-выводе и т. д.). После обсуждения этих основных характеристик мы покажем, как они применяются к операционным системам двух машин из трех наших примеров: Core i7 (Windows 7) и OMAP4430 (Linux). Поскольку контроллер ATmega168 обычно используется для встроенных систем, у этой машины нет операционной системы.

Виртуальная память

В первых компьютерах память была очень мала по объему и к тому же дорого стоила. IBM-650, ведущий компьютер того времени (конец 50-х годов), содержал всего 2000 слов памяти. Один из первых 60 компиляторов ALGOL был написан для компьютера с объемом памяти всего 1024 слова. Древняя система с разделением времени прекрасно работала на компьютере PDP-1, общий объем памяти которого составлял всего 4096 18-разрядных слов для операционной системы и пользовательских программ. В те времена программисты тратили очень много времени, пытаясь вместить свои программы в крошечную память. Часто приходилось использовать более медленный алгоритм только потому, что более быстрый не удавалось разместить в памяти компьютера.

Традиционным решением проблемы было использование вспомогательной памяти (например, диска). Программист делил программу на несколько частей, так называемых **оверлеев**, каждый из которых помещался в память. Чтобы выполнить программу, сначала нужно было считать и запустить первый оверлей. Когда он завершался, считывался и запускался второй оверлей и т. д. Программист отвечал за разбиение программы на оверлеи и решал, в каком месте вспомогательной памяти должен храниться каждый оверлей, контролировал передачу оверлеев между основной и вспомогательной памятью и вообще управлял всем этим процессом без какой-либо помощи со стороны компьютера.

Хотя эта технология широко использовалась на протяжении многих лет, она требовала длительной кропотливой работы, связанной с управлением оверлеями. В 1961 году группа исследователей из Манчестера (Англия) предложила метод автоматической реализации подгрузки кода, при котором программист мог вообще не знать об этом процессе [Fotheringham, 1961]. Этот метод, в основе которого, как сейчас говорят, лежит использование **виртуальной памяти**, имел очевидное преимущество, поскольку освобождал программиста от массы рутинной работы. Впервые этот метод был применен в ряде компьютеров, выпущенных в 60-е годы — в основном для исследовательских проектов в области компьютерных технологий. К началу 70-х годов виртуальная память была реализована

в большинстве компьютеров. В настоящее время даже компьютеры на одной микросхеме, в том числе Core i7 и OMAP4430, содержат очень сложные системы виртуальной памяти. Мы рассмотрим их далее в этой главе.

Страничная организация памяти

Идею о разделении понятий адресного пространства и адресов памяти выдвинула группа ученых из Манчестера. Рассмотрим в качестве примера типичный компьютер того времени с 16-разрядным полем адреса в командах и 4096 словами памяти. Программа, работающая на таком компьютере, могла обращаться к 65 536 словам памяти (поскольку адреса были 16-разрядными, а $2^{16} = 65\,536$). Обратите внимание, что число адресуемых слов зависит только от числа битов адреса и никак не связано с числом реально доступных слов в памяти. **Адресное пространство** такого компьютера состоит из чисел 0, 1, 2, ..., 65 535, так как это — набор всех возможных адресов. Однако в действительности компьютер мог иметь гораздо меньше слов в памяти.

До изобретения виртуальной памяти приходилось проводить жесткое различие между адресами, меньшими 4096, и адресами равными или большими 4096. Эти две части рассматривались как полезное адресное пространство и бесполезное адресное пространство соответственно (адреса выше 4095 были бесполезными, поскольку они не соответствовали реальным адресам памяти). Никакого различия между адресным пространством и адресами памяти не проводилось, поскольку между ними подразумевалось взаимно-однозначное соответствие.

Идея разделения понятий адресного пространства и адресов памяти состоит в следующем. В любой момент времени можно получить прямой доступ к 4096 словам памяти, но это не значит, что они непременно должны соответствовать адресам памяти от 0 до 4095. Например, мы могли бы сообщить компьютеру, что при обращении к адресу 4096 нужно использоваться слово из памяти с адресом 0, при обращении к адресу 4097 — слово из памяти с адресом 1, при обращении к адресу 8191 — слово из памяти с адресом 4095 и т. д. Другими словами, адресное пространство отображается на действительные адреса памяти (рис. 6.2).

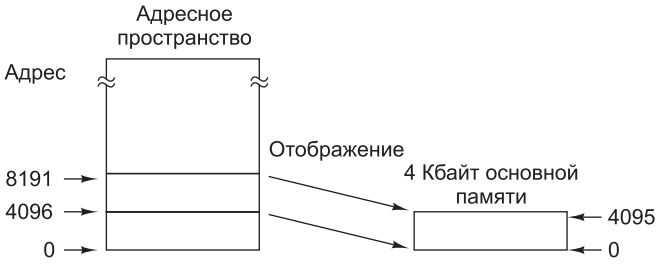


Рис. 6.2. Виртуальные адреса памяти с 4096 по 8191 отображаются на адреса основной памяти с 0 по 4095

Согласно этой схеме отображения адресов адресного пространства на фактические ячейки памяти, в машине с объемом памяти 4 Кбайт (без виртуальной памяти) между адресами от 0 до 4095 и словами памяти числом 4096 существует прямое соответствие. Возникает интересный вопрос: а что произойдет, если про-

грамма совершит переход к одному из адресов в диапазоне от 8192 по 12 287? В машине без виртуальной памяти произойдет ошибка, на экран будет выведено сообщение об обращении к несуществующему адресу памяти, и выполнение программы прервется. На машине с виртуальной памятью произойдет следующее:

1. Содержимое основной памяти будет сохранено на диске.
2. Слова с 8192 по 12 287 будут сохранены на диске.
3. Слова с 8192 до 12 287 будут загружены в основную память.
4. Отображение адресов изменится: адреса с 8192 по 12 287 будут соответствовать ячейкам памяти с 0 по 4095.
5. Выполнение программы продолжится, как будто ничего необычного не случилось.

Такая технология автоматического наложения называется **страничной организацией памяти**, а фрагменты программы, которые считываются с диска, — **страницами**.

Есть и другой, более сложный способ отображения адресов адресного пространства программы на реальные адреса памяти. Адреса, к которым программа может обращаться, мы будем называть **виртуальным адресным пространством**, а реальные адреса памяти, реализованные аппаратно, — **физическим адресным пространством**. В **карте памяти**, или **таблице страниц**, хранится информация о соответствии между виртуальными и физическими адресами. Предполагается, что на диске достаточно места для хранения полного виртуального адресного пространства (или, по крайней мере, той его части, которая используется в данный момент).

Программы пишутся так, как будто в основной памяти хватает места для размещения всего виртуального адресного пространства, даже если это не соответствует действительности. Программы могут загружать слова из виртуального адресного пространства или записывать слова в виртуальное адресное пространство, несмотря на то что на самом деле физической памяти для этого не хватает. Программист может писать программы, даже не зная о существовании виртуальной памяти. Просто с его точки зрения все выглядит так, словно объем памяти данного компьютера достаточно велик.

Позднее мы сравним страничную организацию памяти с процессом сегментации, при котором программисту требуется знать о существовании сегментов. Еще раз подчеркнем, что страничная организация памяти создает иллюзию линейной основной памяти такого же размера, как адресное пространство программы. В действительности основная память может быть меньше (или больше), чем виртуальное адресное пространство. То, что память большого размера просто моделируется путем страничной организации памяти, нельзя определить по программе (только за счет тестирования). При обращении к любому адресу всегда появляются требуемые данные или нужная команда. Поскольку программист может писать программы, ничего не зная о страничной организации памяти, этот механизм называют **прозрачным**.

Ситуация, когда программист использует какой-либо виртуальный механизм и даже не знает, как он работает, не нова. В архитектуры команд, например, часто включается команда `MUL` (умножение), даже если аппаратно умножение не поддерживается. Иллюзия того, что машина может перемножать числа, создается

микропрограммой. Точно так же операционная система может создавать иллюзию, что все виртуальные адреса поддерживаются реальной памятью, даже если это неправда. Только разработчикам и исследователям операционных систем нужно знать, как строится такая иллюзия.

Реализация страничной организации памяти

Виртуальной памяти требуется диск для хранения всей программы и всех данных. Это может быть как традиционное дисковое устройство, так и твердотельный накопитель. Хотя в книге мы будем для простоты использовать термин «диск» или «жесткий диск», следует понимать, что все сказанное в равной степени относится и к твердотельным накопителям. Копию программы, сохраненную на диске, удобнее рассматривать как оригинал, а фрагменты, регулярно записываемые в основную память, — как копии. Естественно, оригинал в таком случае должен оперативно обновляться. Когда изменения вносятся в копию программы в основной памяти, они в конечном итоге должны быть отражены в оригинале.

Виртуальное адресное пространство разбивается на ряд страниц равного размера, обычно от 512 байт до 64 Кбайт, хотя иногда встречаются страницы по 4 Мбайт. Размер страницы всегда должен быть степенью двойки. Физическое адресное пространство тоже разбивается на части равного размера таким образом, чтобы каждая такая часть основной памяти вмещала ровно одну страницу. Эти части основной памяти называются **страничными кадрами**. На рис. 6.2 основная память содержит только один страничный кадр. На практике она обычно состоит из тысяч страничных кадров.

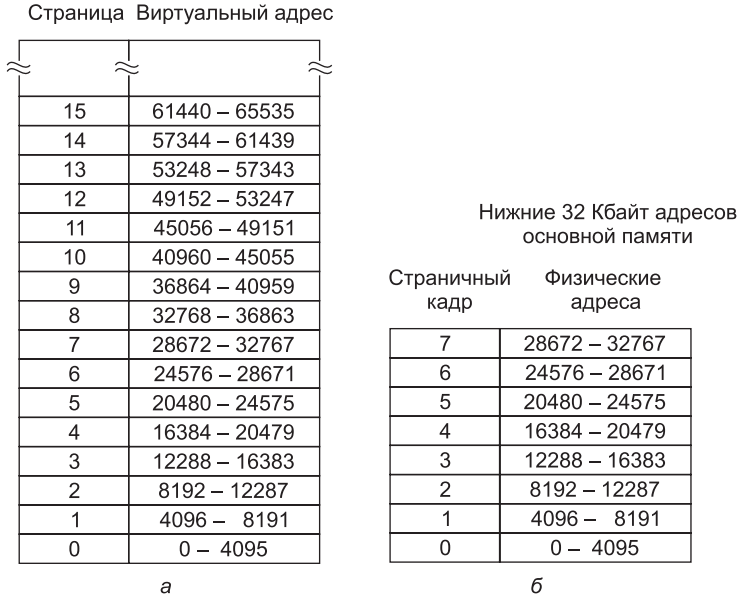


Рис. 6.3. Первые 64 Кбайт виртуального адресного пространства разделены на 16 страниц по 4 Кбайт каждая (*а*); 32 Кбайт основной памяти разделены на 8 страничных кадров по 4 Кбайт каждый (*б*)

На рис. 6.3, *а* показан один из возможных вариантов разделения первых 64 Кбайт виртуального адресного пространства на страницы по 4 Кбайт. Адрес может быть байтом, а может быть словом (например, в компьютере, в котором последовательно расположенные слова имеют последовательные адреса). Виртуальную память, изображенную на рис. 6.3, можно реализовать посредством таблицы страниц, в которой количество элементов равно количеству страниц в виртуальном адресном пространстве. Здесь для простоты мы показали только первые 16 элементов. Когда программа пытается обратиться к слову из первых 64 Кбайт виртуальной памяти, чтобы вызвать команду или данные или чтобы сохранить данные, сначала она генерирует виртуальный адрес от 0 до 65 532 (предполагается, что адреса слов должны делиться на 4). Для этого могут использоваться любые стандартные механизмы адресации, в том числе индексирование и косвенная адресация.

На рис. 6.3, *б* изображена физическая память, состоящая из восьми страничных кадров по 4K. Эту память можно ограничить до 32K, поскольку: 1) это вся память машины (для процессора, встроенного в стиральную машину или микроволновую печь, этого достаточно) или 2) оставшаяся часть памяти занята другими программами.

А теперь посмотрим, как 32-разрядный виртуальный адрес можно отобразить на физический адрес основной памяти. В конце концов, при обращениях к памяти должны использоваться только реальные, а не виртуальные адреса, поэтому без такого отображения не обойтись. Каждый компьютер с виртуальной памятью содержит устройство для отображения виртуальных адресов на физические. Это устройство называется **диспетчером памятью** (Memory Management Unit, **MMU**). Он может находиться на микросхеме процессора или на отдельной микросхеме рядом с процессором. В нашем примере диспетчер памяти отображает 32-разрядный виртуальный адрес на 15-разрядный физический адрес, поэтому ему требуется 32-разрядный входной регистр и 15-разрядный выходной.

Чтобы понять, как работает диспетчер памяти, рассмотрим пример на рис. 6.4. Когда в диспетчер памяти поступает 32-разрядный виртуальный адрес, он разделяет этот адрес на 20-разрядный номер виртуальной страницы и 12-разрядное смещение внутри этой страницы (поскольку страницы в нашем примере имеют размер 4 Кбайт). Номер виртуальной страницы используется в качестве индекса в таблице страниц для нахождения нужной страницы. На рис. 6.4 номер виртуальной страницы равен 3, поэтому в таблице выбирается элемент 3.

Сначала диспетчер памяти проверяет, находится ли нужная страница в текущий момент в памяти. Поскольку у нас есть 2^{20} виртуальных страниц и всего 8 страничных кадров, не все виртуальные страницы могут находиться в памяти одновременно. Диспетчер памяти проверяет **бит присутствия** в данном элементе таблицы страниц. В нашем примере этот бит равен 1. Это значит, что страница в данный момент находится в памяти.

Далее из выбранного элемента таблицы нужно взять значение страничного кадра (в нашем примере — 6) и скопировать его в старшие 3 бита 15-разрядного выходного регистра. Нужно именно 3 бита, потому что в физической памяти находится 8 страничных кадров. Параллельно с этой операцией младшие 12 бит виртуального адреса (поле смещения страницы) копируются в младшие 12 бит

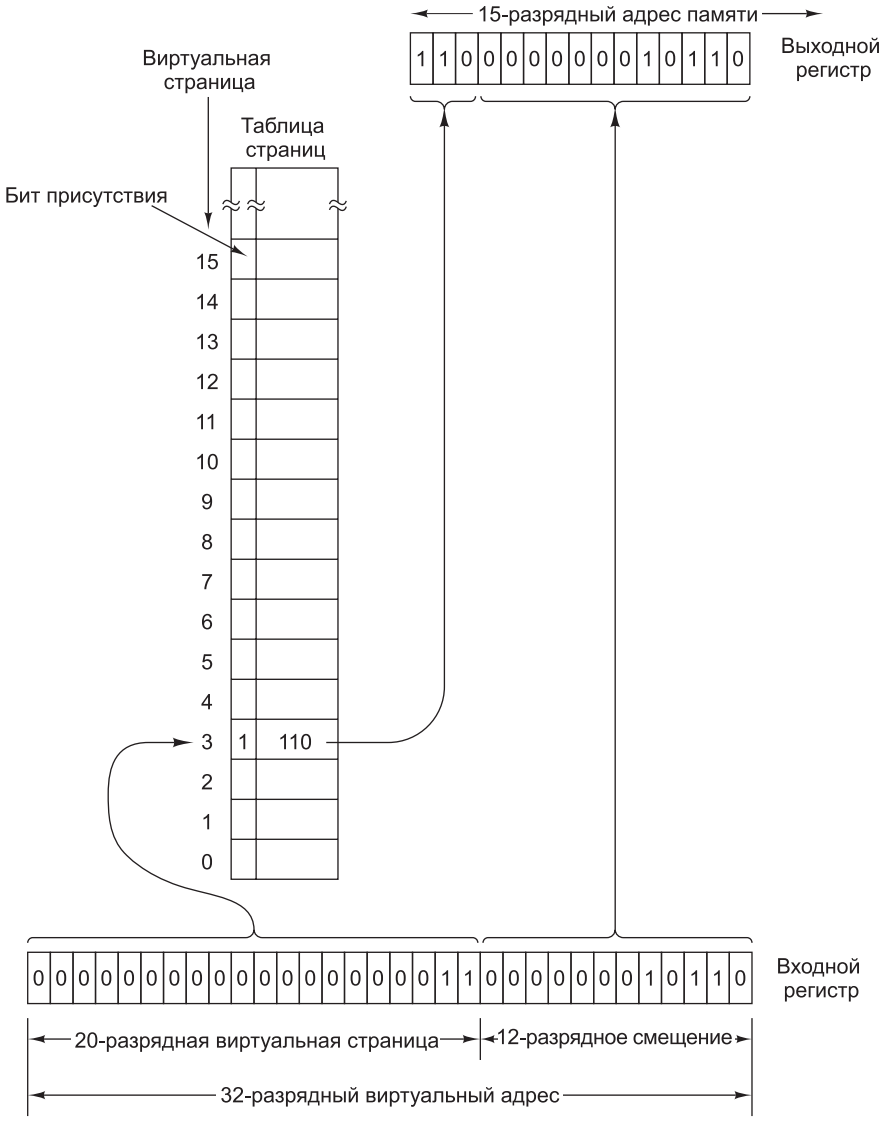


Рис. 6.4. Формирование адреса основной памяти из адреса виртуальной памяти

выходного регистра. Затем полученный 15-разрядный адрес отправляется в кэш-память или основную память для поиска.

На рис. 6.5 показан возможный вариант отображения виртуальных страниц на физические страничные кадры. Виртуальная страница 0 находится в страничном кадре 1. Виртуальная страница 1 находится в страничном кадре 0. Виртуальной страницы 2 нет в основной памяти. Виртуальная страница 3 находится в страничном кадре 2. Виртуальной страницы 4 нет в основной памяти. Виртуальная страница 5 находится в страничном кадре 6 и т. д.

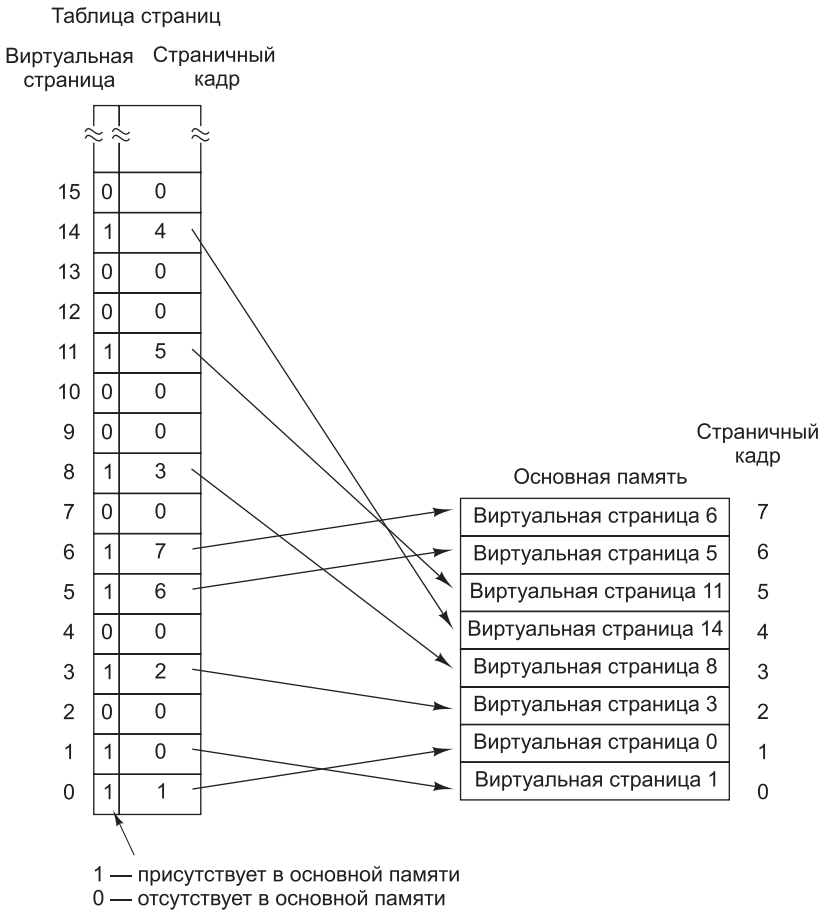


Рис. 6.5. Возможное отображение первых 16 виртуальных страниц в основную память, содержащую 8 страничных кадров

Вызов страниц по требованию и рабочее множество

В предыдущем обсуждении предполагалось, что виртуальная страница, к которой происходит обращение, находится в основной памяти. Однако это предположение не всегда верно, поскольку в основной памяти недостаточно места для всех виртуальных страниц. При обращении к адресу страницы, которой нет в основной памяти, происходит **ошибка отсутствия страницы**. В случае такой ошибки операционная система должна считать нужную страницу с диска, ввести новый адрес физической памяти в таблицу страниц, а затем повторить команду, вызвавшую ошибку.

На машине с виртуальной памятью можно запустить программу даже в том случае, если в основной памяти нет ни одной части программы. Просто в таблице страниц содержится информация о том, что абсолютно все виртуальные страницы находятся во вспомогательной памяти. Если центральный процессор попытается вызвать первую команду, он сразу получит ошибку отсутствия

страницы, в результате чего страница, содержащая первую команду, будет загружена в память и внесена в таблицу страниц. После этого начнется выполнение первой команды. Если первая команда содержит два адреса и оба эти адреса находятся на разных страницах, причем не на той, в которой находится команда, то произойдут еще две ошибки отсутствия страницы и еще две страницы будут перенесены в основную память до завершения команды. Следующая команда может вызвать еще несколько ошибок и т. д.

Такой метод работы с виртуальной памятью называется **вызовом страниц по требованию** (demand paging). Такой вызов страниц напоминает кормление младенца по требованию (в противоположность кормлению по расписанию): когда младенец кричит, вы его кормите. При вызове страниц по требованию страницы переносятся в основную память только в случае необходимости, но не ранее.

Вопрос о том, стоит ли использовать механизм вызова по требованию или нет, имеет смысл только при запуске программы. Когда программа проработает некоторое время, нужные страницы уже окажутся в основной памяти. Однако если это компьютер с разделением времени и процессы выгружаются, проработав примерно 100 миллисекунд, то каждая программа будет запускаться многократно. Для каждой программы распределение памяти уникально и при переключении с одной программы на другую меняется, поэтому в системах с разделением времени такой подход не годится.

Альтернативный подход основан на наблюдении, что многие программы обращаются к адресному пространству неравномерно. Обычно большинство обращений относятся к небольшому числу страниц. Это называется **принципом локальности**. При обращении к памяти можно вызвать команду, вызвать данные или сохранить данные. В каждый момент времени t существует набор страниц, которые использовались при последних k обращениях. В [Denning, 1968] этот набор страниц назван **рабочим множеством**.

Поскольку рабочее множество обычно меняется очень медленно, можно, опираясь на последнее перед остановкой программы рабочее множество, предсказать, какие страницы понадобятся при новом запуске программы. Эти страницы можно загрузить заранее перед очередным запуском программы (если они, конечно, поместятся в памяти).

Политика замещения страниц

В идеале, чтобы сократить количество ошибок отсутствия страниц, можно хранить в памяти набор страниц, которые постоянно используются программой (так называемое рабочее множество). Однако программисты обычно не знают, какие страницы находятся в рабочем множестве, поэтому операционная система должна сама определять это множество динамически. Если программа обращается к странице, которая отсутствует в основной памяти, ее нужно вызвать с диска. Однако чтобы освободить для нее место, на диск нужно отправить какую-нибудь другую страницу. Следовательно, нужен алгоритм, позволяющий определить, какую именно страницу необходимо удалить из памяти.

Выбирать такую страницу просто наугад нельзя. Если, например, выбрать страницу, содержащую команду, выполнение которой вызвало ошибку, то при попытке вызвать следующую команду произойдет еще одна ошибка отсутствия страницы.

Большинство операционных систем стараются предсказать, какие из страниц в памяти наименее полезны в том смысле, что их отсутствие значительно не повлияет на ход программы. Один из методов решения этой задачи таков: сначала прогнозируется время следующих обращений к каждой странице, а затем страница, обращение к которой, согласно прогнозу, должно произойти позже всех, удаляется. Иными словами, вместо того чтобы удалять страницу, которая скоро будет нужна, стараются выбрать такую страницу, которая не понадобится достаточно долго.

По одному из алгоритмов удаляется та страница, которая дольше других не использовалась, поскольку вероятность того, что она окажется в текущем рабочем множестве, очень мала. Этот алгоритм называется **LRU** (Least Recently Used — **дольше всего не использовавшийся**). Хотя этот алгоритм работает достаточно хорошо, в некоторых патологических ситуациях он дает сбой. Вот одна из них.

Представьте себе программу, выполняющую огромный цикл, занимающий 9 виртуальных страниц, в то время как в физической памяти место есть только для восьми страниц. Когда программа перейдет к странице 7, в основной памяти будут находиться страницы с 0 по 7. Затем совершается попытка вызвать команду с виртуальной страницы 8, что вызывает ошибку отсутствия страницы. В соответствии с алгоритмом LRU из памяти удаляется виртуальная страница 0, поскольку она не использовалась дольше других. Виртуальная страница 0 удаляется, а на ее место помещается виртуальная страница 8 (в памяти оказываются страницы 1–8). Ситуация выглядит так, как показано на рис. 6.6.

Рис. 6.6. Ситуация, в которой алгоритм LRU не работает

После выполнения команд с виртуальной страницы 8 программа возвращается к началу цикла, то есть к виртуальной странице 0. Этот шаг вызывает еще одну ошибку отсутствия страницы, только что выброшенную виртуальную страницу 0 приходится вызывать обратно. В соответствии с алгоритмом LRU из памяти удаляется страница 1 (в памяти оказываются страница 0 плюс страницы 2–8). Через некоторое время программа пытается вызвать команду с виртуальной страницы 1, что опять вызывает ошибку. Затем вызывается страница 1 и удаляется страница 2 и т. д.

Очевидно, что в этой ситуации алгоритм LRU совершенно не работает (другие алгоритмы при сходных обстоятельствах тоже не работают). Однако если объем доступной памяти превышает размер рабочего множества, число ошибок отсутствия страниц станет минимальным.

Можно применить другой алгоритм — **FIFO** (First-in First-out — **первым пришел, первым ушел**). В соответствии с алгоритмом FIFO удаляется та страница, которая была загружена раньше всех, независимо от того, когда в последний раз производилось обращение к этой странице. С каждым страничным кадром связан отдельный счетчик. Изначально все счетчики сброшены. После каждой ошибки отсутствия страницы счетчик каждой страницы, находящейся в памяти, увеличивается на единицу, а счетчик только что вызванной страницы сбрасывается. Для удаления выбирается страница с самым большим значением счетчика. Поскольку страница не загружалась в память очень давно, существует большая вероятность, что она больше не понадобится.

Если размер рабочего множества больше, чем число доступных страничных кадров, ни один алгоритм не дает хороших результатов, и ошибки отсутствия страниц происходят часто. Если программа постоянно вызывает подобные ошибки, то говорят, что наблюдается **пробуксовка** (thrashing). Вероятно, не нужно объяснять, почему пробуксовка нежелательна. Если программа использует большое виртуальное адресное пространство, но имеет компактное и медленно меняющееся рабочее множество, которое помещается в основную память, все отлично работает. Это утверждение имеет силу, даже если программа использует в сотни раз больше слов виртуальной памяти, чем их содержится в физической памяти.

Если удаляемая страница не менялась с тех пор, как была считана (а это вполне вероятно, если страница содержит программу, а не данные), записывать ее обратно на диск не обязательно, поскольку точная копия там уже есть. Однако если страница менялась, ее копия на диске ей уже не соответствует, и страницу нужно сбросить на диск.

Если бы существовала возможность определять, изменялось ли содержимое страницы с момента чтения, это позволило бы избежать ненужных операций записи на диск и сэкономить много времени. На многих машинах в диспетчере памяти для каждой страницы выделятся 1 бит, который сбрасывается при загрузке страницы и устанавливается, когда эта страница меняется (микропрограммой или аппаратно). По этому биту операционная система определяет, менялась данная страница или нет и нужно ее сбрасывать на диск или нет.

Размер страниц и фрагментация

Если пользовательская программа и данные занимают ровно целое число страниц, то когда они время от времени загружаются в память, свободного места там не остается. В то же время, если они не занимают ровно целое число страниц, на последней странице останется неиспользованное пространство. Например, если программа и данные занимают 26 000 байт на машине со страницами по 4096 байт, то первые 6 страниц будут заполнены целиком, что в сумме даст $6 \times 4096 = 24\,576$ байт, а последняя страница будет содержать $26\,000 - 24576 = 1424$ байта. Поскольку в каждой странице имеется пространство для 4096 байт, 2672 байта останутся свободными. Всякий раз, когда страница 7 будет оказываться в памяти, эти байты останутся неиспользуемыми, бесполезно занимая место в основной памяти. Эта проблема получила название **внутренней фрагментации** (поскольку неиспользуемое пространство является внутренним по отношению к странице).

Если размер страницы составляет n байт, то среднее неиспользуемое пространство последней страницы программы будет равно $n/2$ байт. Таким образом, чтобы свести к минимуму объем бесполезного пространства, страницы должны быть небольшими. Однако при маленьких страницах потребуется много страниц и большая таблица страниц. Если таблица страниц сохраняется аппаратно, то для хранения большой таблицы страниц нужно много регистров, что повышает стоимость компьютера. Кроме того, при запуске и остановке программы на загрузку и сохранение этих регистров потребуется больше времени.

Больше того, маленькие страницы снижают эффективность использования пропускной способности диска. Поскольку перед началом передачи данных с диска приходится ждать примерно 10 миллисекунд (время поиска плюс задержка вращения), выгодно передать побольше данных. При скорости передачи данных 100 Мбайт/с, передача 8 Кбайт по сравнению с передачей 1 Кбайт добавляет всего 70 микросекунд.

Однако у маленьких страниц есть свои преимущества. Если рабочее множество состоит из большого количества маленьких, отделенных друг от друга областей виртуального адресного пространства, то при небольшом размере страницы реже будет возникать пробуксовка (режим интенсивной подкачки), чем при большом. Рассмотрим матрицу A размером 10 000 × 10 000, которая хранится в последовательных 8-байтных словах (A[1,1], A[2,1], A[3,1] и т. д.). При такой записи элементы строки 1 (A[1,1], A[1,2], A[1,3] и т. д.) будут начинаться на расстоянии 80 000 байт друг от друга. Программе, обрабатывающей элементы этой строки, потребуется 10 000 областей, каждая из которых отделена от следующей величиной 79 992 байт. Если бы размер страницы составлял 8 Кбайт, для хранения всех страниц понадобилось бы 80 Мбайт.

При размере страницы в 1 Кбайт для хранения всех страниц потребуется всего 10 Мбайт ОЗУ. При размере памяти в 32 Мбайт и размере страницы в 8 Кбайт программа войдет в режим интенсивной подкачки, а при размере страницы в 1 Кбайт этого не произойдет. С учетом всех факторов сейчас имеется тенденция к увеличению размеров страниц. На практике 4 Кбайт сегодня считается минимальной величиной.

Сегментация

До сих пор мы говорили об одномерной виртуальной памяти, в которой виртуальные адреса следуют один за другим от 0 до какого-то максимального адреса. Однако по многим причинам гораздо удобнее использовать два или несколько отдельных виртуальных адресных пространств. Например, компилятор может иметь несколько структур, которые создаются в процессе компиляции; такими структурами могут быть:

- таблица символических имен с именами и атрибутами переменных;
- исходный текст для распечатки;
- таблица всех используемых целочисленных констант и констант с плавающей точкой;
- дерево синтаксического разбора программы;
- стек, используемый для вызова процедур в компиляторе.

Каждая из первых четырех структур постоянно растет в процессе компиляции. Размер последней структуры меняется совершенно непредсказуемо. В одномерной памяти эти пять структур пришлось бы разместить в виртуальном адресном пространстве в виде смежных областей, как показано на рис. 6.7.

Посмотрим, что произойдет, если программа содержит очень большое число переменных. Область адресного пространства, предназначенная для таблицы символов, может переполниться, даже если в других таблицах полно свободного места. Компилятор, конечно, может сообщить, что он не способен продолжать работу из-за большого количества переменных, но можно без этого и обойтись, поскольку в других таблицах много свободного места.

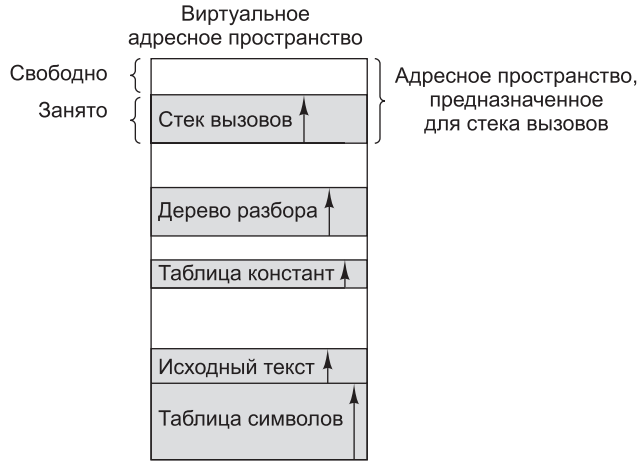


Рис. 6.7. Если в одномерном адресном пространстве размер структур постоянно растет, одна из них может «врезаться» в другую

Компилятор может забирать свободное пространство у одних структур и передавать другим, но это будет напоминать управление оверлеями вручную — некоторое неудобство в лучшем случае и долгая нудная работа в худшем.

На самом деле нужно просто освободить программиста от необходимости думать о расширении и сокращении структур, подобно тому, как виртуальная память позволяет забыть о разбиении программы на оверлеи.

Для этого нужно создать множество абсолютно независимых адресных пространств, которые называются **сегментами**. Каждый сегмент состоит из линейной последовательности адресов от 0 до какого-либо допустимого максимума. Разные сегменты могут иметь разную длину (обычно так и бывает). Более того, длина сегмента может меняться во время выполнения программы. Длина стекового сегмента может увеличиваться всякий раз, когда что-либо помещается в стек, и уменьшаться, когда что-либо выталкивается из стека.

Так как каждый сегмент образует отдельное адресное пространство, разные сегменты могут увеличиваться или уменьшаться независимо друг от друга и не влияя друг на друга. Если стеку в определенном сегменте потребуется больше адресного пространства (чтобы стек мог расти), он может получить его, поскольку его адресному пространству больше не во что «врезаться». Естественно, сегмент может переполниться, но это происходит редко, поскольку сегменты очень

большие. Чтобы задать адрес в двухмерной памяти, программа должна указать номер сегмента и адрес внутри сегмента. Сегментированная память изображена на рис. 6.8.

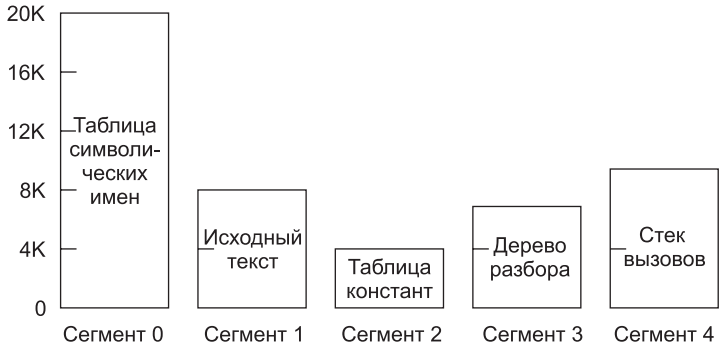


Рис. 6.8. Сегментированная память позволяет менять размер каждой структуры независимо от других

Подчеркнем, что сегмент является *логическим* элементом, о котором программист знает и который он использует. Сегмент может содержать процедуру, массив, стек или ряд скалярных переменных, но обычно в него входит только тип содержимого.

Сегментированная память имеет и другие преимущества, помимо упрощения работы со структурами данных, которые постоянно меняются в размерах. Если каждая процедура занимает отдельный сегмент, в котором первый адрес — это адрес 0, то сборка процедур, скомпилированных раздельно, значительно упрощается. Когда все процедуры программы скомпилированы и собраны, при вызове процедуры из сегмента n для обращения к слову 0 будет использоваться адрес (n, 0).

Если процедура в сегменте n впоследствии будет изменена и перекомпилирована, то другие процедуры менять не потребуется (поскольку их начальные адреса не изменятся), даже если новая версия окажется больше по размеру, чем предыдущая. В одномерной памяти процедуры обычно располагаются друг за другом, и между ними нет свободного адресного пространства. Следовательно, изменение размера одной процедуры может повлиять на начальные адреса других процедур. Это, в свою очередь, потребует изменения всех других процедур, которые вызывают любую из указанных. Если программа содержит сотни процедур, переделка оказывается весьма трудоемкой.

Сегментация облегчает совместное использование процедур или данных несколькими программами. Если компьютер содержит несколько программ, работающих параллельно (это может быть реальный или моделируемый параллелизм), и все эти программы вызывают одни и те же процедуры, снабжать каждую программу отдельными копиями процедур расточительно для памяти. Если же сделать каждую процедуру отдельным сегментом, их легко можно будет использовать совместно, что исключит необходимость создавать физические копии каждой разделяемой процедуры. В результате достигается экономия памяти.

Поскольку каждый сегмент являет собой логический объект (например, процедуру, массив или стек), о существовании которого знает программист, разные сегменты можно защищать по-разному. Например, сегменту с процедурой можно назначить атрибут «только для выполнения», запретив тем самым считывание из него и запись в него. Для массива с плавающей точкой можно разрешить только чтение и запись, но не выполнение и т. д. Такая защита часто помогает выявлять программные ошибки.

Нужно понимать, почему защита имеет смысл только для сегментированной памяти и не имеет никакого смысла для одномерной (линейной) памяти. Обычно в сегменте не могут находиться одновременно процедура и стек (только что-нибудь одно). Поскольку каждый сегмент содержит объект только одного типа, он может использовать защиту, подходящую для этого типа. Страничная организация памяти и сегментация сравниваются в табл. 6.1.

Таблица 6.1. Сравнение страничной организации памяти и сегментации

Свойство	Страничная организация памяти	Сегментация
Необходимость учета программистом	Нет	Да
Количество линейных адресных пространств	1	Много
Возможность увеличения памяти для виртуального адресного пространства	Да	Да
Простота управления структурами переменного размера	Нет	Да
Назначение технологии	Имитация памяти большого размера	Предоставление нескольких адресных пространств

Содержимое страницы в каком-то смысле произвольно. Программист может ничего не знать о страничной организации памяти. В принципе можно поместить несколько битов в каждый элемент таблицы страниц для доступа к нему, но чтобы использовать эту особенность, программисту придется следить за границами страницы в адресном пространстве. Но ведь механизм разбиения памяти на страницы был придуман именно для устранения подобных трудностей. Что же касается сегментации, то поскольку у пользователя создается иллюзия, что все сегменты постоянно находятся в основной памяти и к ним можно обращаться без хлопот, связанных с управлением оверлеями.

Реализация сегментации

Сегментацию можно реализовать одним из двух способов: подкачкой сегментов и разбиением памяти на страницы. При первом подходе в памяти находится некоторый набор сегментов. Если происходит обращение к сегменту, которого нет в памяти, этот сегмент переносится в память. Если для него нет места в памяти, один или несколько сегментов сначала сбрасываются на диск (если на диске уже есть их копия, сегменты просто удаляется из памяти). В каком-то смысле подкачка сегментов очень похожа на вызов страниц по требованию: сегменты загружаются и удаляются по мере необходимости.

Однако сегментация существенно отличается от разбиения на страницы в следующем: размер страниц фиксирован, а размер сегментов — нет. На рис. 6.9, *а* показан пример физической памяти, в которой изначально содержится 5 сегментов. Посмотрим, что происходит, если сегмент 1 удаляется, а сегмент 7, который меньше по размеру, помещается на его место. В результате получается конфигурация, изображенная на рис. 6.9, *б*. Между сегментом 7 и сегментом 2 оказывается неиспользованная («пустая») область. Затем сегмент 4 заменяется сегментом 5 (рис. 6.9, *в*), а сегмент 3 — сегментом 6 (рис. 6.9, *г*). Через некоторое время память разделится на ряд областей, одни из которых будут содержать сегменты, а другие — пусто́ты. Это называется **внешней фрагментацией** (поскольку неиспользованное пространство попадает не в сегменты, а в пустоты между ними, то есть процесс происходит вне сегментов). Иногда внешнюю фрагментацию называют **поклеточной разбивкой** (checkerboarding).

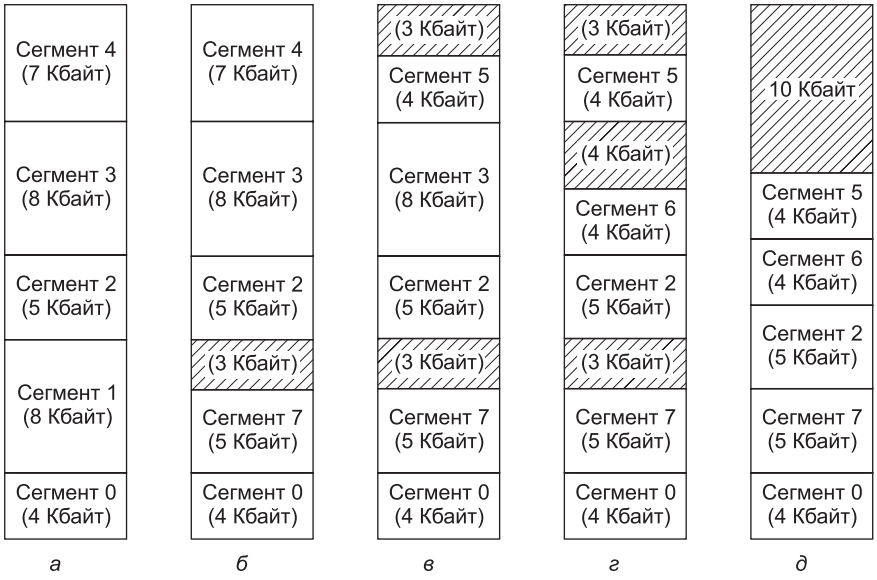


Рис. 6.9. Динамика внешней фрагментации (*а*, *б*, *в*, *г*); дефрагментация путем уплотнения (*д*)

Посмотрим, что произойдет, если программа обратится к сегменту 3, когда память фрагментирована (см. рис. 6.9, *г*). Общее пространство пустот составляет 10 Кбайт, и хотя это больше, чем нужно для сегмента 3, сегмент 3 туда не помещается, так как пространство разбито на маленькие фрагменты. Вместо этого приходится сначала удалять другой сегмент.

Один из способов предотвращения подобной ситуации заключается в следующем: каждый раз при появлении пустого пространства следующие сегменты перемещаются ближе к адресу 0, удаляя таким образом это пустое пространство (точнее, «сдвигая» его к концу памяти). Есть и другой способ. Можно подождать, пока внешняя фрагментация не станет серьезно влиять на процессы (когда на долю пустот придется больше некоторого процента от всего объема памяти), и только после этого выполнить уплотнение. На рис. 6.8, *д* показано, как память

будет выглядеть после уплотнения. Цель уплотнения памяти — собрать все маленькие пустоты в одно большое свободное пространство, в которое поместится один или несколько сегментов. Недостаток уплотнения состоит в том, что на этот процесс тратится некоторое время.

Если на уплотнение памяти требуется слишком много времени, нужен специальный алгоритм для определения того, какие именно пустоты лучше использовать для определенного сегмента. Для этого требуется список адресов и размеров всех пустот. В популярном алгоритме **оптимальной подгонки** (best fit) выбирается самая маленькая из пустот, в которую помещается нужный сегмент. Цель этого алгоритма — не допустить использования большого свободного фрагмента, который может понадобиться позже для большого сегмента.

В другом популярном алгоритме список пустот просто просматривается по кругу и выбирается первый же свободный фрагмент, по размеру подходящий для данного сегмента. Удивительно, но последний алгоритм с точки зрения общей производительности работает гораздо лучше, чем алгоритм оптимальной подгонки, поскольку последний порождает очень много маленьких пустот [Knuth, 1997].

Оба алгоритма сокращают средний размер свободного фрагмента. Всякий раз, когда сегмент помещается в свободный фрагмент, превышающий по размеру этот сегмент, что бывает практически всегда (точные попадания очень редки), свободный фрагмент делится на две части. Одну часть занимает сегмент, а вторая часть превращается в новый свободный фрагмент, который по определению меньше прежнего. Без дефрагментации (превращения всех маленьких пустот в одну большую) оба алгоритма в конечном итоге приведут к заполнению памяти пустотами, слишком маленькими даже для самого мелкого сегмента.

Опишем один из таких процессов. Всякий раз, когда сегмент удаляется из памяти, а одна или обе соседние области этого сегмента — не сегменты, а пустоты, смежные пустоты можно объединить. Если на рис. 6.8, *г* удалить сегмент 5, то две соседние пусто́ты, объединившись с фрагментом размером 4 Кбайт, который занимал удаленный сегмент, дадут в результате свободный фрагмент размером уже 11 Кбайт.

В начале этого подраздела мы говорили о том, что реализовать сегментацию можно двумя способами: подкачкой сегментов и разбиением на страницы. До сих пор речь шла о подкачке. При таком подходе по необходимости между памятью и диском перемещаются целые сегменты. Второй способ реализации — разделить каждый сегмент на страницы фиксированного размера и вызывать их по требованию. В этом случае одни страницы сегмента могут находиться в памяти, а другие — на диске. Чтобы разбить сегмент на страницы, для каждого сегмента нужна отдельная таблица страниц. Поскольку сегмент представляет собой линейное адресное пространство, все варианты разбиения на страницы, которые мы до сих пор рассматривали, применимы к любому сегменту. Единственное отличие состоит в том, что каждый сегмент получает отдельную таблицу страниц.

MULTICS (Multiplexed Information and Computing Service — **служба общей информации и вычислений**) — это древняя операционная система, совмещающая сегментацию с разбиением на страницы. Она была разработана Массачусетским технологическим институтом совместно с компаниями Bell Labs и General Electric [Corbat and Vyssotsky, 1965; Organick, 1972]. Адреса в MULTICS состоят из двух частей: номера сегмента и адреса внутри сегмента. Для каждого про-

цесса существовал дескрипторный сегмент, содержащий дескрипторы каждого сегмента. При аппаратном получении виртуального адреса для нахождения дескриптора нужного сегмента номер сегмента использовался в качестве индекса в дескрипторном сегменте (рис. 6.10). Дескриптор указывал на таблицу страниц, что позволяло разбивать на страницы каждый сегмент обычным способом. Для повышения производительности недавно использованные сочетания сегментов и страниц помещались в **ассоциативную память** из 16 элементов. Операционная система MULTICS уже давно не применяется, но прожила она очень долго — с 1965 года до 30 декабря 2000 года, когда был отключен последний компьютер с системой MULTICS. Редкая операционная система живет 35 лет. Более того, ее дух продолжает жить, поскольку виртуальная память всех процессоров Intel, начиная с 386, очень похожа на эту систему. Описания истории и других аспектов MULTICS можно найти на сайте *www.multicians.org*.

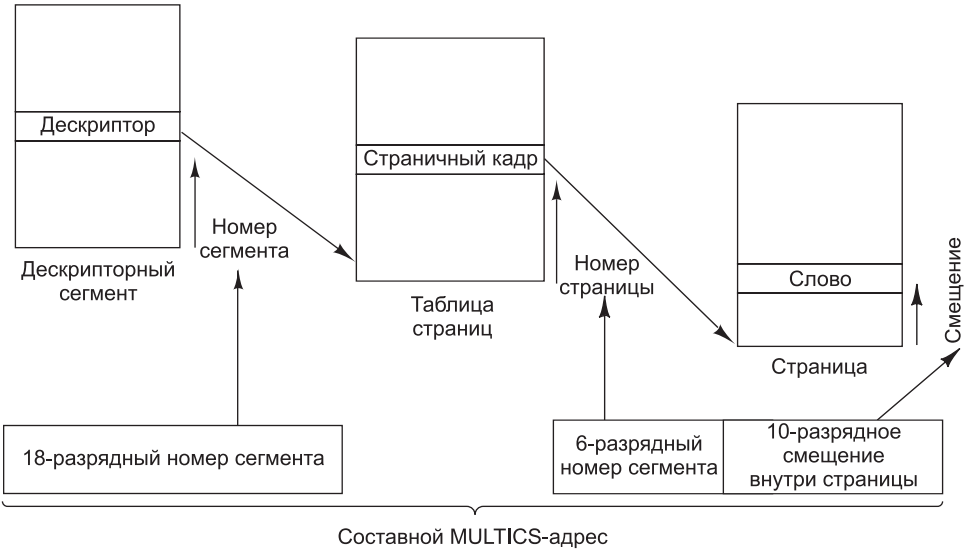


Рис. 6.10. Преобразование составного MULTICS-адреса в адрес основной памяти

Виртуальная память Core i7

Core i7 имеет сложную систему виртуальной памяти, которая поддерживает вызов страниц по требованию, чистую сегментацию и сегментацию с разбиением на страницы. Виртуальная память состоит из двух таблиц: **LDT** (Local Descriptor Table — **локальная таблица дескрипторов**) и **GDT** (Global Descriptor Table — **глобальная таблица дескрипторов**). Каждая программа имеет собственную локальную таблицу дескрипторов, а единственная глобальная таблица дескрипторов разделяется всеми программами компьютера. Локальная таблица дескрипторов описывает локальные сегменты каждой программы (ее код, данные, стек и т. д.), а глобальная таблица дескрипторов — системные сегменты, в том числе самой операционной системы.

Как уже отмечалось в главе 5, чтобы получить доступ к сегменту, Core i7 сначала загружает селектор сегмента в один из сегментных регистров. Во время

выполнения программы регистр CS содержит селектор сегмента кода, DS — селектор сегмента данных и т. д. Каждый селектор представляет собой 16-разрядное число (рис. 6.11).

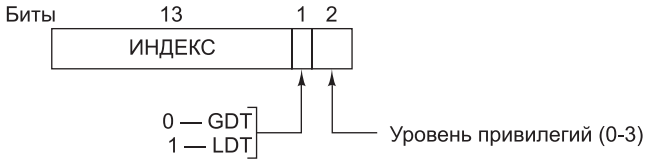


Рис. 6.11. Селектор Core i7

Один из битов селектора показывает, является сегмент локальным или глобальным (то есть к какой из двух таблиц дескрипторов, локальной или глобальной, он относится). Еще 13 бит определяют номер элемента в локальной или глобальной таблице дескрипторов, поэтому объем каждой из этих таблиц ограничен значением 8 Кбайт (2^{13}) сегментных дескрипторов. Оставшиеся два бита связаны с защитой. Мы опишем их позже. Дескриптор 0 недействителен и вызывает исключение. Его можно загрузить в регистр сегмента, чтобы показать, что регистр сегмента недоступен, но если попытаться использовать дескриптор 0, будет вызвано исключение.

Когда селектор загружается в сегментный регистр, соответствующий дескриптор вызывается из локальной или глобальной таблицы дескрипторов и сохраняется во внутренних регистрах диспетчера памяти, поэтому к нему можно быстро получить доступ. Дескриптор состоит из 8 байт. Сюда входит базовый адрес сегмента, его размер и другая информация (рис. 6.12).

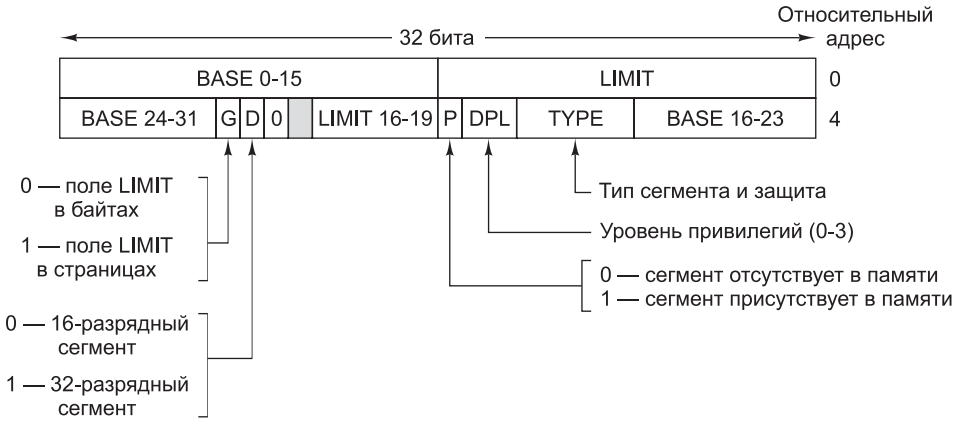


Рис. 6.12. Дескриптор сегмента кода Core i7.
Сегменты данных практически ничем не отличаются

Формат селектора выбран таким образом, чтобы упростить поиск дескриптора. Сначала на основе бита 2 в селекторе выбирается локальная или глобальная таблица дескрипторов. Затем селектор копируется во временный регистр диспетчера памяти, а три младших бита принимают значение 0 — в результате 13-разрядное число селектора умножается на 8. Наконец, к этому значению прибавляется адрес из локальной или глобальной таблицы дескрипторов (который

хранится во внутренних регистрах диспетчера памяти), и в результате получается указатель на дескриптор. Например, селектор 72 обращается к элементу 9 в глобальной таблице дескрипторов, который находится в ячейке с адресом GDT + 72.

Давайте проследим, каким образом пара (селектор, смещение) превращается в физический адрес. Как только аппаратное обеспечение определяет, какой именно регистр сегмента используется, оно находит во внутренних регистрах полный дескриптор, соответствующий данному селектору. Если такого сегмента не существует (селектор 0) или в данный момент он не находится в памяти (P = 0), вызывается исключение. В первом случае это — программная ошибка; второй случай требует, чтобы операционная система вызвала нужный сегмент.

Затем аппаратное обеспечение проверяет, не выходит ли смещение за пределы сегмента. Если выходит, то снова вызывается исключение. По логике вещей в дескрипторе должно быть 32-разрядное поле для определения размера сегмента, но там в наличии имеется всего 20 бит, поэтому в данном случае используется совершенно другая схема. Если поле G (Granularity — степень детализации) равно 0, то поле LIMIT (максимальное значение) дает точный размер сегмента (до 1 Мбайт). Если поле G равно 1, то поле LIMIT указывает размер сегмента в страницах, а не в байтах. Размер страницы в компьютере Core i7 никогда не бывает меньше 4 Кбайт, поэтому 20 бит достаточно для сегментов размером до 2^{32} байт.

Если сегмент находится в памяти, а смещение не выходит за границу сегмента, Core i7 прибавляет 32-разрядное поле BASE (базовый адрес) в дескрипторе к смещению, в результате чего получается **линейный адрес** (рис. 6.13). Поле BASE разбивается на три части и разносится по дескриптору, чтобы обеспечить совместимость с процессором 80286, в котором размер BASE составляет всего 24 бита. Поэтому каждый сегмент может начинаться с произвольного места в 32-разрядном адресном пространстве.

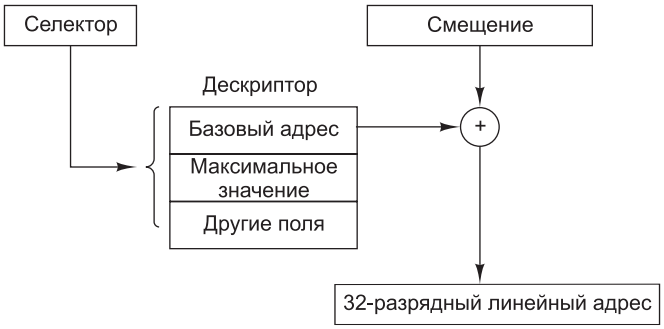


Рис. 6.13. Преобразование пары селектор-смещение в линейный адрес

Если разбиение на страницы блокировано (это определяется по биту в глобальном регистре управления), линейный адрес интерпретируется как физический адрес и отправляется в память для чтения или записи. Таким образом, при блокировке разбиения на страницы мы имеем «чистую» схему сегментации, где каждый базовый адрес сегмента присутствует в его дескрипторе. Допускается перекрытие сегментов, поскольку было бы слишком утомительно тратить много времени на проверку того, что сегменты непересекающиеся.

Если же разбиение на страницы разрешено, линейный адрес интерпретируется как виртуальный и отображается на физический адрес с использованием таблиц страниц, почти как в наших примерах. Единственная сложность состоит в том, что при 32-разрядном виртуальном адресе и 32-разрядных страницах 4-килобайтный сегмент может содержать один миллион страниц, поэтому для сокращения размера таблицы страниц при маленьких сегментах применяется двухуровневое отображение.

Каждая работающая программа имеет специальную **таблицу страниц**, которая состоит из 1024 32-разрядных элементов. Ее адрес указывается глобальным регистром. Каждый элемент в этой таблице указывает на таблицу страниц, которая также содержит 1024 32-разрядных элементов. Элементы таблицы страниц указывают на страничные кадры. Схема изображена на рис. 6.14.

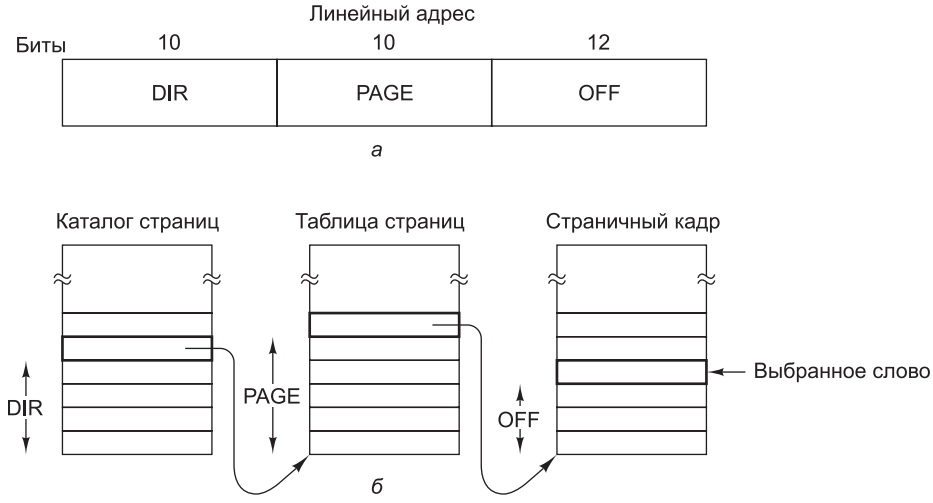


Рис. 6.14. Отображение линейного адреса на физический

На рис. 6.14, *а* мы видим линейный адрес, разбитый на три поля: DIR, PAGE и OFF. Поле DIR используется в качестве индекса в каталоге страниц для нахождения указателя на нужную таблицу страниц. Поле PAGE является индексом в таблице страниц при нахождении физического адреса страничного кадра. Наконец, поле OFF прибавляется к адресу страничного кадра для формирования физического адреса нужного байта или слова.

Размер каждого элемента таблицы страниц — 32 бита, 20 из которых содержат номер страничного кадра. Оставшиеся биты включают бит доступа и бит изменения, которые устанавливаются аппаратно, чтобы помочь операционной системе, биты защиты и некоторые другие.

В каждой таблице страниц содержатся элементы для 1024 страничных кадров по 4 Кбайт каждый, поэтому одна таблица страниц может работать со значением 4 Мбайт памяти. Сегмент короче 4 Мбайт будет иметь каталог страниц с одним элементом (указателем на его единственную таблицу страниц). Таким образом, непроизводительные издержки для коротких сегментов составляют всего две страницы, а не миллион страниц, как было бы в одноуровневой таблице страниц.

Чтобы избежать повторных обращений к памяти, диспетчер памяти Core i7 имеет встроенную аппаратную поддержку для поиска недавно использовавшихся комбинаций полей DIR-PAGE и отображения их на физический адрес соответствующего страничного кадра. Действия, показанные на рис. 6.14, выполняются только в том случае, если текущая комбинация использовалась давно.

При разбиении на страницы значение поля BASE в дескрипторе вполне может быть нулевым. Единственное, для чего нужно поле BASE, — получить небольшое смещение, чтобы использовать элемент в середине, а не в начале каталога страниц. Поле BASE включено в дескриптор только для реализации чистой сегментации (без разбиения на страницы), а также для обратной совместимости со старым процессором 80286, в котором не было разбиения на страницы.

Отметим, что если конкретное приложение не нуждается в сегментации и довольствуется единым 32-разрядным адресным пространством со страничной организацией, этого легко достичь. Все сегментные регистры могут быть заполнены одним и тем же селектором, дескриптор которого содержит нулевое поле BASE и максимальное поле LIMIT. Смещение команды будет тогда линейным адресом с единственным адресным пространством, то есть, по сути, получается традиционное разбиение на страницы.

На этом наше рассмотрение организации виртуальной памяти Core i7 подходит к концу. Мы рассмотрели небольшую (но часто используемую) подсистему виртуальной памяти Core i7; любознательный читатель может обратиться к документации Core i7 за информацией о расширениях 64-разрядной адресации и поддержке виртуализованных физических адресных пространств. Однако в завершение стоит сказать несколько слов о защите, поскольку это имеет непосредственное отношение к виртуальной памяти. Core i7 поддерживает 4 уровня защиты, где уровень 0 — самый привилегированный, а уровень 3 — наименее привилегированный (рис. 6.15). Уровень защиты работающей программы указывается 2-разрядным полем в **слове состояния программы** (Program Status Word, **PSW**) — регистре аппаратного обеспечения, который содержит коды условий

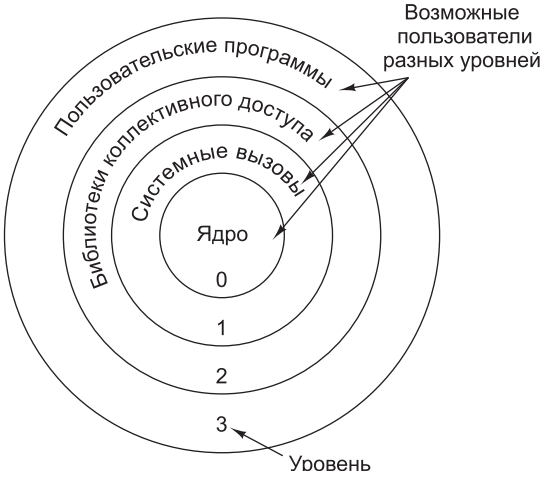


Рис. 6.15. Уровни защиты процессора Core i7

и другие биты состояния. Более того, не только программы, но и каждый сегмент в системе имеет определенный уровень защиты.

Пока программа использует сегменты только собственного уровня, все идет нормально. Доступ к данным более высокого уровня разрешается. Доступ к данным более низкого уровня запрещен — в этом случае происходит исключение. Допустим вызов процедур как более высокого, так и более низкого уровня, но при этом нужно строго контролировать ситуацию. Для вызова процедуры другого уровня команда `CALL` должна содержать селектор вместо адреса. Этот селектор указывает на дескриптор, который называется **шлюзом вызова** (call gate) и по которому можно получить адрес нужной процедуры. Таким образом, совершить переход в середину произвольного сегмента на другом уровне невозможно. Могут использоваться только официальные точки входа.

Посмотрите на рис. 6.15. На уровне 0 мы видим ядро операционной системы, которая контролирует процесс ввода-вывода, работу памяти и т. п. На уровне 1 находится обработчик системных вызовов. Пользовательские программы могут вызывать процедуры из этого уровня, но это только строго определенные процедуры. Уровень 2 содержит библиотечные процедуры, которые могут использоваться совместно несколькими работающими программами. Пользовательские программы могут вызывать эти процедуры, но не могут изменять их. На уровне 3 работают пользовательские программы, которые имеют самую низкую степень защиты. Система защиты Core i7, как и схема управления памятью, в целом основана на принципах системы MULTICS.

В исключениях и прерываниях используется похожий механизм. Исключения и прерывания тоже обращаются к дескрипторам, а не к абсолютным адресам, а эти дескрипторы указывают на процедуры, которые нужно выполнить. Поле TYPE на рис. 6.11 позволяет различать сегменты кода, сегменты данных и разнообразные логические элементы.

Виртуальная память OMAP4430

OMAP4430 — 32-разрядная машина, которая поддерживает виртуальную память со страничной организацией на базе 32-разрядных адресов, преобразуемых в 32-разрядное физическое адресное пространство. Соответственно, процессор ARM способен поддерживать до 2^{32} (4 Гбайт) физической памяти. Поддерживаются четыре варианта размера страниц: 4 Кбайт, 64 Кбайт, 1 Мбайт и 16 Мбайт. Механизм отображения для этих четырех размеров страниц показаны на рис. 6.16.

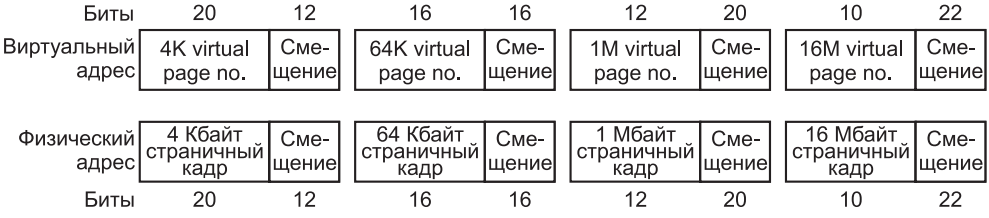


Рис. 6.16. Отображение виртуальных адресов на физические в OMAP4430

Структура страничных таблиц OMAP4430 очень похожа на структуру Core i7. Механизм отображения для 4-килобайтных страниц показан на рис. 6.17 *а*. Таблица дескрипторов первого уровня индексируется старшими 12 битами виртуального адреса. Элемент таблицы дескрипторов первого уровня определяет физический адрес таблицы дескрипторов второго уровня. Этот адрес, в сочетании со следующими 8 битами виртуального адреса, формирует адрес дескриптора страницы. Дескриптор страницы содержит адрес кадра физической страницы и информацию, определяющую права доступа к странице.

Механизм виртуальной памяти OMAP4430 поддерживает четыре размера страниц. Страницы с размерами 1 Мбайт и 16 Мбайт отображаются на дескрипторы страниц, находящиеся в таблице дескрипторов первого уровня. Необходимость в таблицах второго уровня в данном случае отсутствует, поскольку все элементы указывают на одну большую физическую страницу. Дескрипторы 64-килобайтных страниц находятся в таблице второго уровня. Так как каждый элемент таблицы дескрипторов второго уровня связывает 4-килобайтную виртуальную страницу с 4-килобайтной физической страницей, для страницы с размером 64 Кбайт в таблице дескрипторов второго уровня должны находиться 16 идентичных дескрипторов. Почему программист ОС, находящийся в здравом уме, будет объявлять страницу с размером 64 Кбайт, когда можно использовать более гибкие страницы по 4 Кбайт? Потому что, как мы вскоре увидим, страницы с размером 64 Кбайт требуют меньше элементов TLB, а этот ресурс критичен для хорошей производительности.

Ничто не замедляет работу программы сильнее, чем «узкие места» при работе с памятью. Внимательно присмотревшись к рис. 6.17, вы увидите, что каждое обращение к памяти программы требует двух дополнительных обращений к памяти для преобразования адреса. 200 % дополнительных затрат на каждое обращение к памяти, необходимые для преобразования виртуальных адресов, способны затормозить работу любой программы. Для устранения этого фактора OMAP4430 использует так называемый **буфер быстрого преобразова-**

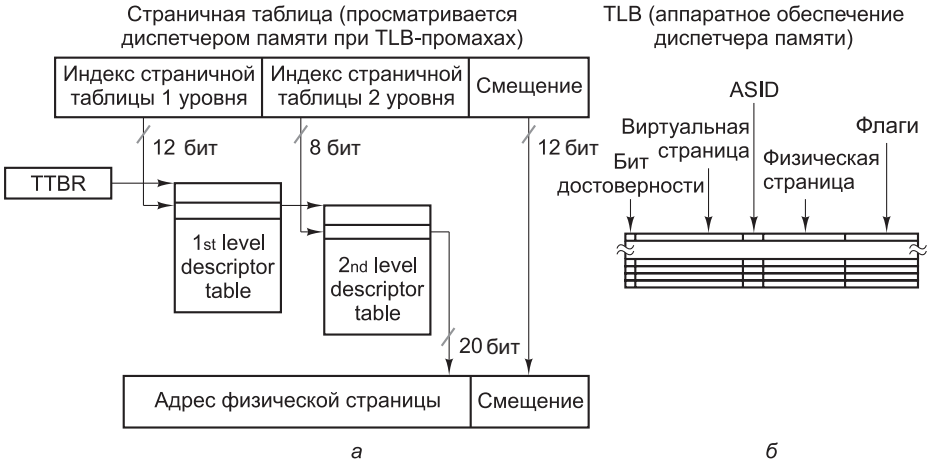


Рис. 6.17. Структуры данных, использующиеся для трансляции виртуального адреса OMAP4430: таблица преобразования адресов (*а*); буфер TSB (*б*)

ния (Translation Lookaside Buffer, TLB) для быстрого преобразования номеров виртуальных страниц и номеров физических страничных кадров. Для страниц размером в 4 Кбайт существуют 2^{20} номеров виртуальных страниц, то есть более миллиона. Естественно, не все они могут быть отображены.

В TLB хранятся только номера только тех виртуальных страниц, которые использовались последними. Страницы команд и данных рассматриваются отдельно. Для каждой из этих категорий в TLB включены номера 128 последних виртуальных страниц. Каждый элемент TLB включает номер виртуальной страницы и соответствующий ему номер физического страничного кадра. Когда номер процесса, называемый **ASID** (Address Space Identifier), и виртуальный адрес передаются диспетчеру памяти, диспетчер с помощью специальной схемы одновременно сравнивает номер виртуальной страницы сразу со всеми элементами буфера TLB для данного контекста. Если обнаруживается совпадение, номер страничного кадра в этом элементе буфера соединяется со смещением, взятом из виртуального адреса, для получения 32-разрядного физического адреса и некоторых флагов (например, битов защиты). Буфер быстрого преобразования адреса изображен на рис. 6.17, *б*.

Если совпадение не обнаруживается, то есть имеет место **TLB-промах** (по аналогии с кэш-промахом), происходит аппаратный перебор страничных таблиц. Когда в страничной таблице обнаруживается новый элемент дескриптора физической страницы, он проверяется на нахождение страницы в памяти, и если проверка дает положительный результат, соответствующее преобразование адреса загружается в TLB. Если страница не находится в памяти, инициируется стандартная ошибка отсутствия страницы. Так как количество элементов в TLB невелико, вероятность вытеснения существующего элемента TLB достаточно велика. Будущим обращениям к вытесненной странице придется снова перебирать содержимое страничных таблиц для получения отображения адреса. При слишком быстром вытеснении слишком многих страниц начнется пробуксовка, и большинство обращений к памяти будет требовать 200 % дополнительных затрат.

Интересно сравнить организацию виртуальной памяти в Core i7 и OMAP4430. Core i7 поддерживает «чистую» сегментацию, «чистое» разбиение на страницы и сегментацию в сочетании с разбиением на страницы. OMAP4430 поддерживает только разбиение на страницы. И Core i7, и OMAP4430 в случае TLB-промаха перезагружают элемент буфера TLB аппаратно. Другие архитектуры (например, SPARC и MIPS) просто передают управление операционной системе. В этих архитектурах определяются специальные привилегированные команды для работы с TLB, чтобы операционная система могла выполнить операции просмотра страничных таблиц и загрузки TLB, необходимые для преобразования адресов.

Виртуальная память и кэширование

На первый взгляд может показаться, что виртуальная память и кэширование никак не связаны, но на самом деле эти механизмы похожи. При наличии виртуальной памяти вся программа хранится на диске и разбивается на страницы фиксированного размера. Некоторое подмножество этих страниц находится в основной памяти. Если программа главным образом использует страницы

из основной памяти, то ошибки отсутствия страницы будут встречаться редко, и программа сможет работать быстро. При кэшировании вся программа хранится в основной памяти и разбивается на блоки фиксированного размера. Некоторое подмножество этих блоков находится в кэш-памяти. Если программа главным образом использует блоки из кэш-памяти, то кэш-промахи будут происходить редко, и программа сможет работать быстро. Как видим, виртуальная память и кэш-память идентичны, разве что работают они на разных уровнях иерархии.

Естественно, виртуальная память и кэш-память кое в чем различаются. Кэш-промахи обрабатываются аппаратно, а ошибки отсутствия страниц обрабатываются операционной системой. Блоки кэш-памяти обычно гораздо меньше страниц (сравните: 64 байта против 8 Кбайт). Кроме того, таблицы страниц индексируются по старшим битам виртуального адреса, а кэш-память индексируется по младшим битам адреса памяти. Тем не менее важно понимать, что различие здесь только в реализации, а на концептуальном уровне существует значительное сходство.

Виртуализация оборудования

Традиционно аппаратные архитектуры проектировались с расчетом на то, что на них в любой момент времени выполняется только одна операционная система. Популярные ныне технологии совместного доступа к вычислительным ресурсам (например, серверам облачных вычислений) сильно выигрывают от возможности одновременной работы нескольких операционных систем. Например, поставщики услуг интернет-хостинга обычно предоставляют своим клиентам полноценный доступ к системе, на базе которой строится веб-сервис. Было бы слишком расточительно устанавливать в серверной новый компьютер для каждого нового клиента. Вместо этого обычно применяется **виртуализация**, обеспечивающая возможность выполнения нескольких полноценных систем (включая операционную систему) на одном сервере. Только когда нагрузка на существующие серверы становится слишком высокой, в пул серверов добавляется новый физический сервер.

Чисто программные технологии виртуализации существуют, но обычно они замедляют работу виртуальной системы и требуют особых модификаций операционной системы или использования сложных анализаторов кода для изменения программ в процессе выполнения. Все это привело к тому, что проектировщики расширили уровень операционной системы в своих архитектурах, чтобы обеспечить эффективную поддержку виртуализации непосредственно на аппаратном уровне.

Виртуализация оборудования (рис. 6.18) представляет собой совокупность аппаратных и программных средств, обеспечивающих возможность одновременного выполнения нескольких операционных систем на одном физическом компьютере. С точки зрения конечного пользователя, каждая виртуальная машина. работающая на физическом компьютере, выглядит как автономная операционная система. Созданием и управлением экземплярами виртуальных машин занимается **гипервизор** — программный компонент, имеющий много общего с ядром операционной системы. Аппаратное обеспечение поставляет события, видимые на программном уровне, и необходимые гипервизору для реализации совместного доступа к процессору, системе хранения данных и устройствам ввода-вывода.

Рис. 6.18. Аппаратная поддержка виртуализации позволяет одновременно выполнять несколько операционных систем на одном физическом компьютере. Гипервизор управляет совместным использованием памяти и устройств ввода-вывода

Существование на одном физическом компьютере нескольких виртуальных машин, на которых выполняются разные операционные системы, открывает много полезных возможностей. В серверных системах виртуализация позволяет системным администраторам создать несколько виртуальных машин на одном физическом сервере и перемещать работающие виртуальные машины между серверами для оптимального распределения общей нагрузки. Виртуальные машины также предоставляют средства для более точного управления доступом к устройствам ввода-вывода. Например, пропускная способность виртуализированного сетевого порта может распределяться в зависимости от уровней обслуживания пользователей. Отдельному пользователю виртуализация дает возможность одновременного выполнения нескольких операционных систем.

Чтобы виртуализация могла быть реализована на аппаратном уровне, все команды архитектуры должны обращаться только к ресурсам текущей виртуальной машины. Для большинства команд это требование реализуется элементарно. Например, арифметические команды должны работать только с регистровым файлом, который может быть виртуализирован посредством копирования регистров виртуальной машины в регистровый файл процессора при переключении контекста.

С виртуализацией команд обращения к памяти (например, команд загрузки и сохранения) дело обстоит сложнее, так как эти команды должны обращаться только к ячейкам физической памяти, выделенным текущей виртуальной машине. Как правило, процессор с поддержкой аппаратной виртуализации должен предоставить дополнительный механизм отображения страниц, который связывает страницы физической памяти виртуальной машины со страницами физической памяти компьютера. Наконец, команды ввода-вывода (включая ввод-вывод, отображаемый на память) не должны напрямую работать с физическими устройствами ввода-вывода, так как политика виртуализации часто подразумевает распределение доступа к устройствам ввода-вывода. Детализированный контроль ввода-вывода обычно реализуется прерываниями к гипервизору каждый раз, когда виртуальная машина пытается обратиться к устройству ввода-вывода. Это позволяет гипервизору реализовать такую политику доступа к ресурсам ввода-вывода, какую он считает нужным. Обычно поддерживается набор устройств ввода-вывода, и операционные системы, выполняемые на виртуальных машинах (**гостевые операционные системы**), используют только эти поддерживаемые устройства.

Аппаратная виртуализация в Core i7

Аппаратная виртуализация в Core i7 поддерживается расширениями виртуальных машин **VMX** (Virtual Machine eXtensions) — совокупностью расширений команд, памяти и прерываний, обеспечивающих эффективное управление виртуальными машинами. В VMX виртуализация памяти реализуется системой **EPT** (Extended Page Table), которая может использоваться при аппаратной виртуализации. EPT преобразует адреса физических страниц виртуальной машины в физические адреса компьютера при помощи дополнительной многоуровневой табличной структуры, содержимое которое перебирается при TLB-промахе виртуальной машины. Таблица поддерживается гипервизором, который может организовать любую нужную схему совместного использования физической памяти.

Виртуализация операций ввода-вывода (как команд, так и ввода-вывода, отображаемого на память) реализуется на базе расширенной поддержки прерываний, определенной в **VMCS** (Virtual-Machine Control Structure). Прерывание гипервизора вызывается каждый раз, когда виртуальная машина обращается к устройству ввода-вывода. При получении прерывания гипервизор реализует операцию ввода-вывода на программном уровне по правилам, необходимым для совместного использования устройств ввода-вывода виртуальными машинами.

Виртуальные команды ввода-вывода

Как вам уже известно, наборы команд на уровне архитектуры команд и на уровне микроархитектуры совершенно разные. Различаются не только сами команды, но и их форматы, а некоторые совпадения совершенно случайны.

Набор команд уровня операционной системы содержит большую часть команд уровня архитектуры команд, а также несколько новых очень важных команд. В то же время некоторые новые, но важные команды на уровне операционной системы не поддерживаются. Ввод-вывод — это одна из областей, в которых эти два уровня отличаются очень значительно. Причина таких различий проста. Во-первых, пользователь, способный выполнять команды ввода-вывода уровня архитектуры команд, может считать конфиденциальную информацию, хранящуюся где-нибудь в «дебрях» операционной системы, что потенциально представляет для системы угрозу. Во-вторых, нормальный программист совершенно не горит желанием реализовать ввод-вывод на уровне архитектуры команд, поскольку это слишком сложно и утомительно. Вместо этого для ввода-вывода можно просто установить определенные поля и биты в ряде регистров устройств, затем подождать, пока операция закончится, и проверить, что произошло. Обычно биты регистров дисковых устройств позволяют обнаруживать следующие ошибки:

- аппаратура диска не смогла выполнить позиционирование;
- несуществующий элемент памяти определен как буфер;
- процесс ввода-вывода с диска (на диск) начался до того, как закончился предыдущий;
- ошибка синхронизации при считывании;
- обращение к несуществующему диску;
- обращение к несуществующему цилиндру;

- обращение к несуществующему сектору;
- несоответствие контрольных сумм при считывании;
- ошибка проверки записи.

При наличии одной из этих ошибок устанавливается соответствующий бит в регистре устройства. Мало кому из пользователей захочется отслеживать все эти ошибки вместе с большим объемом информации состояния.

Файлы

Один из способов виртуального ввода-вывода — использование абстракции под названием **файл**. Файл состоит из последовательности байтов, записанных на устройство ввода-вывода. Если устройство ввода-вывода является устройством хранения информации (например, диском), то файл можно считать обратно. Если устройство не является устройством хранения информации (например, это принтер), то файл оттуда считать нельзя. На диске может храниться множество файлов, в каждом из которых содержатся данные определенного типа, например изображение, электронная таблица или текст. Файлы имеют разную длину и обладают разными свойствами. Эта абстракция позволяет легко организовать виртуальный ввод-вывод.

Для операционной системы файл является просто последовательностью байтов. Вся структуризация определяется исключительно на уровне прикладных программ. Ввод-вывод осуществляется путем системных вызовов открытия, чтения, записи и закрытия файлов. Перед тем как считывать файл, его нужно открыть. Процесс открытия файла позволяет операционной системе найти файл на диске и передать в память информацию, необходимую для доступа к этому файлу.

После открытия файла из него можно читать данные. Системный вызов для считывания должен иметь, как минимум, следующие параметры:

- информацию о том, какой именно открытый файл нужно считывать;
- указатель на буфер в памяти, в который нужно поместить данные;
- число считываемых байтов.

Данный системный вызов помещает требующиеся данные в буфер. Обычно он возвращает число считанных байтов. Это число может быть меньше запрошенного числа (например, нельзя считать 2000 байт из файла размером 1000 байт).

С каждым открытым файлом связан указатель, который сообщает, какой байт должен считываться следующим. После команды `read` указатель дополняется числом считанных байтов, поэтому последовательные команды `read` считывают последовательные блоки данных из файла. Обычно этот указатель можно установить на особое значение, чтобы программы могли получить доступ к любой части файла. Когда программа заканчивает считывание файла, она может закрыть его и сообщить операционной системе, что больше не будет использовать этот файл. Тогда операционная система сможет освободить пространство в структуре, в которой хранилась информация об этом файле.

Мэйнфреймы все еще используются (особенно для обслуживания очень крупных сайтов электронной коммерции), и на многих из них работают традиционные операционные системы (хотя многие работают под управлением Linux).

В традиционных операционных системах мэйнфреймов используется несколько иная модель файла. Здесь файл может быть последовательностью **логических записей**, каждая из которых имеет строго определенную структуру. Например, логическая запись может представлять собой структуру данных, состоящую из пяти полей: двух символьных строк, «Имя» и «Начальник», двух целых чисел, «Отдел» и «Комната», и одного логического значения «ПолЖенский». Некоторые операционные системы различают файлы, в которых все элементы имеют одинаковую структуру, и файлы, содержащие разные типы данных.

Основная виртуальная команда ввода считывает следующую запись из нужного файла и помещает ее в основную память, начиная с определенного адреса, как показано на рис. 6.19. Чтобы выполнить эту операцию, виртуальная команда должна получить сведения о том, какой файл считывать и куда в памяти поместить запись. Часто существуют параметры для чтения некоторой записи, которые определяются либо по ее месту в файле, либо по ее ключу.

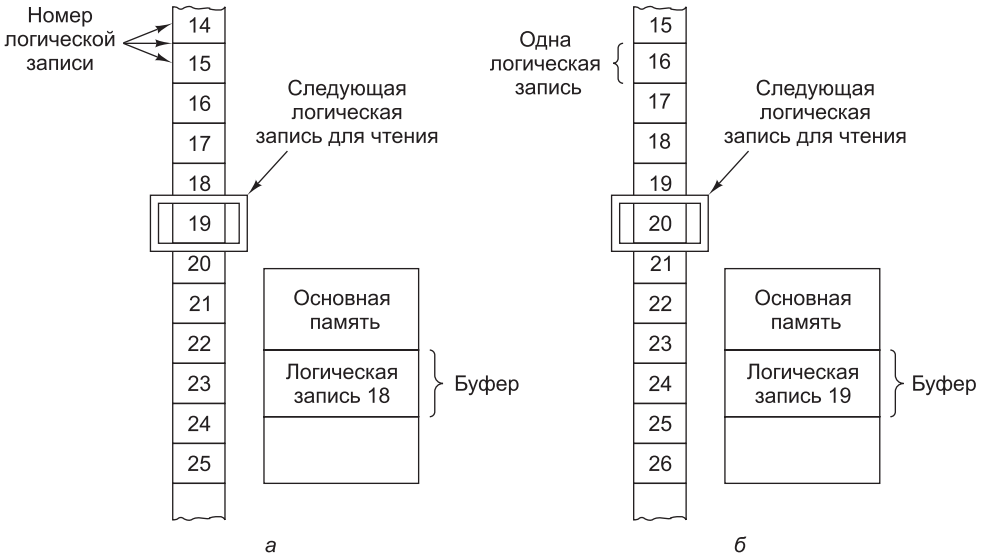


Рис. 6.19. Чтение файла, состоящего из логических записей: до чтения записи 19 (*а*); после чтения записи 19 (*б*)

Основная виртуальная команда вывода записывает логическую запись из памяти в файл. Последовательные команды `write` выполняют последовательные логические записи в файл.

Реализация виртуальных команд ввода-вывода

Чтобы понять, как реализуются виртуальные команды ввода-вывода, нужно хорошо понимать механизмы структурирования и хранения файлов. Основная проблема, которая должна быть решена для всех файловых систем, — распределение пространства. Единицей распределения (также называемой «блоком») может быть один сектор на диске, но чаще блок состоит из нескольких последовательных секторов.

Еще одно фундаментальное свойство реализации файловой системы — то, как располагаются блоки размещения (последовательно или нет). На рис. 6.20 изображен простой диск с одной поверхностью, состоящий из пяти дорожек по 12 секторов каждая. На рис. 6.18, *а* файл состоит из последовательных секторов. Последовательное расположение блоков характерно для компакт-дисков. На рис. 6.18, *б* файл занимает непоследовательные секторы. Такая схема традиционна для жестких дисков (и конечно, для твердотельных накопителей).

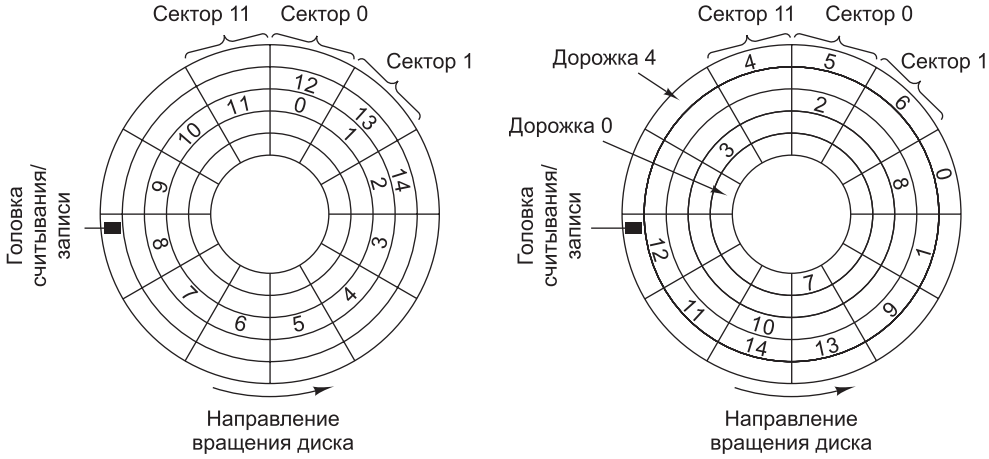


Рис. 6.20. Варианты расположения файла на диске: файл занимает последовательные секторы (*а*); файл занимает непоследовательные секторы (*б*)

Восприятие файла прикладным программистом значительно отличается от восприятия файла операционной системой. Программист воспринимает файл как линейную последовательность байтов или логических записей. Операционная система воспринимает файл как упорядоченную, хотя необязательно последовательную, совокупность распределяемых блоков на диске.

Чтобы операционная система по запросу могла предоставить байт или логическую запись *n* из какого-то файла, она должна использовать какой-либо метод для определения местонахождения данных. Если файл расположен последовательно, для вычисления позиции нужного байта или логической записи операционная система должна знать только место начала файла.

Если файл расположен на диске не последовательно, то лишь по начальной позиции файла невозможно вычислить позицию произвольного байта или логической записи в этом файле. Чтобы найти произвольный байт или логическую запись, нужна таблица, называемая **файловым индексом** и позволяющая получать хранящиеся на диске блоки и их физические адреса. Индекс может быть организован либо в виде списка адресов блоков (такая схема используется в UNIX), либо в виде списка логических записей, для каждой из которых даются адрес на диске и смещение. Иногда каждая логическая запись имеет **ключ**, и программы могут обращаться к записи по этому ключу, а не по номеру логической записи. В последнем случае каждый элемент таблицы должен содержать не

только информацию о местонахождении записи на диске, но и ее ключ. Подобная структура обычно применяется в мэйнфреймах.

Альтернативный метод поиска распределяемых блоков файла — организация файла в виде связного списка. В этом случае каждый блок размещения содержит адрес следующего блока. Для эффективной реализации этой схемы в основной памяти хранится таблица со всеми последующими адресами. Например, для диска с 64-килобайтными блоками операционная система может иметь в памяти таблицу из 64 Кбайт элементов, в каждом из которых есть индекс следующего блока размещения. Так, если файл занимает блоки 4, 52 и 19, то элемент 4 в таблице будет содержать число 52, элемент 52 — число 19, а элемент 19 — специальный код, который указывает на конец файла (например, 0 или –1). Так работали файловые системы в MS-DOS, Windows 95 и Windows 98. Более новые версии Windows (2000, XP, Vista и 7) поддерживают эту файловую систему, но имеет также собственную, которая больше похожа на файловую систему UNIX.

До сих пор мы говорили, что на диске файлы могут располагаться как последовательно, так и непоследовательно, но зачем нужны эти два варианта расположения, мы еще не объяснили. Последовательными файлами легко управлять, но если максимальный размер файла заранее не известен, использовать эту технологию нельзя. Если файл начинается с сектора j и продолжается в направлении соседних секторов, он может наткнуться на другой файл в секторе k, тогда ему не хватит места на расширение. Если же файл располагается непоследовательно, то таких проблем не возникает, поскольку следующие блоки можно поместить в другое место на диске. Если диск содержит ряд «рабочих» файлов, конечные размеры которых могут меняться, записать их на диск в последовательные блоки размещения практически невозможно. Иногда можно переместить существующий файл, но это всегда связано со значительными затратами времени и системных ресурсов.

В то же время, если максимальный размер всех файлов известен заранее (например, как бывает при записи компакт-диска), программа может заранее выделить последовательности секторов, точно равных по длине каждому файлу. Если файлы размером 1200, 700, 2000 и 900 секторов нужно поместить на компакт-диск, они просто могут начинаться с секторов 0, 1200, 1900 и 3900 соответственно (оглавление здесь не учитывается). Найти любую часть любого файла легко, поскольку известен первый сектор файла.

Чтобы выделить пространство на диске для файла, операционная система должна следить, какие блоки доступны, а какие уже заняты другими файлами. При записи на компакт-диск вычисление производится один раз и навсегда, а на жестком диске файлы постоянно записываются и удаляются. Один из способов отслеживать состояние диска — хранить список всех пустот (неиспользованных фрагментов), где пустой фрагмент может быть размером с любое число последовательных блоков размещения. Этот список называется **списком свободной памяти** (free list). На рис. 6.21, *а* изображен список свободной памяти для диска с рисунка 6.20, *б*.

Альтернативное решение — хранить битовую карту диска (один бит на один распределяемый блок), как показано на рис. 6.21, *б*. Единичный бит показывает, что блок уже занят, а нулевой — что свободен.

Дорожка	Сектор	Число секторов в неиспользуемом фрагменте
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

а

	Сектор											
Дорожка	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

б

Рис. 6.21. Два подхода к трассировке свободных секторов: список свободной памяти (*а*); битовая карта (*б*)

Первый подход позволяет легко найти неиспользуемый фрагмент определенной длины. Однако у этого метода есть недостаток: по мере создания и уничтожения файлов размер списка будет меняться, а это нежелательно. Преимущество битовой карты состоит в том, что ее размер постоянен. Кроме того, для изменения статуса блока размещения (из свободного на занятый или наоборот) достаточно поменять значение всего одного бита. Однако при таком подходе трудно найти блок заданного размера. Оба метода требуют, чтобы при записи файла на диск или удаления файла с диска список размещения или карта обновлялись.

Перед тем как закончить обсуждение вопроса о реализации файловой системы, нужно сказать несколько слов о размере блока размещения. Здесь играют роль несколько факторов. Во-первых, тормозят доступ к диску время поиска и время, затрачиваемое на вращение диска. Если на нахождение начала блока тратится 10 миллисекунд, то гораздо выгоднее считать 8 Кбайт (это займет примерно одну миллисекунду), чем 1 Кбайт (это займет примерно 0,125 миллисекунду), так как если считывать 8 Кбайт как 8 блоков по 1 Кбайт, потребуется выполнять поиск 8 раз. Для повышения производительности нужны большие блоки размещения. Конечно, по мере снижения цены и роста популярности твердотельных накопителей этот аргумент постепенно утрачивает свою важность, потому что у этих устройств время позиционирования равно нулю.

Также существует еще один довод в пользу больших блоков: чем меньше размер блока, тем больше их должно быть. Большое количество распределяемых блоков, в свою очередь, влечет за собой большие файловые индексы и большие структуры связных списков в памяти. Например, в системе MS-DOS изначально размер блока составлял один сектор (512 байт), а сектора идентифицировались 16-разрядными числами. Когда размер дисков стал превышать 64 Кбайт секторов, использовать все дисковое пространство с 16-разрядной идентификацией блоков можно было только одним способом: увеличением размера блока. В первом выпуске Windows 95 существовала та же проблема, но в последующем выпуске уже использовались 32-разрядные числа. Windows 98 поддерживала оба варианта.

Впрочем, маленькие блоки тоже имеют свои преимущества. Дело в том, что файлы очень редко занимают ровно целое число блоков размещения. Следовательно, практически в каждом файле в последнем блоке размещения останется неиспользованное пространство. Если размер файла значительно превышает размер блока размещения, то в среднем неиспользованное пространство составит половину блока. Чем больше блок, тем больше остается свободного пространства. Если средний размер файла намного меньше размера блока размещения, большая часть пространства на диске окажется неиспользованной.

Например, в MS-DOS или в первой версии Windows 95 с 2-гигабайтным дисковым разделом размер блока размещения составляет 32 Кбайт, поэтому при записи на диск файла в 100 символов 32 668 байт дискового пространства теряются. С точки зрения распределения дискового пространства маленькие блоки размещения имеют преимущество над большими. В настоящее время самым важным фактором считается скорость передачи данных, поэтому размер блоков постоянно увеличивается.

Команды управления каталогами

Много лет назад программы и данные хранились на перфокартах. Поскольку размер программ увеличивался, а данных становилось все больше, такая форма хранения стала неудобной. Тогда возникла идея вместо перфокарт использовать для хранения программ и данных вспомогательную память (например, диск). Информация, доступная для компьютера без вмешательства человека, называется **оперативной** (on-line). Напротив, **автономная** (off-line) информация требует вмешательства человека (например, чтобы вставить компакт-диск, USB-накопитель или карту SD).

Оперативная информация хранится в файлах. Программы могут получить доступ к ней через программы ввода-вывода. Чтобы следить за оперативной информацией, группировать ее в удобные блоки и защищать от незаконного использования, нужны дополнительные команды.

Обычно операционная система группирует файлы оперативной информации в **каталоги**. На рис. 6.22 проиллюстрирован пример такой организации. В этом случае поддерживаются системные вызовы, по крайней мере, для следующих функций:

- создание файла и включение его в каталог;
- удаление файла из каталога;
- переименование файла;
- изменение статуса защиты файла.

Все современные операционные системы позволяют хранить более одного каталога. Каждый каталог обычно сам представляет собой файл и, как таковой, может включаться в другой каталог, в результате получается иерархическая древовидная структура каталогов. Большое количество каталогов особенно удобно программистам, работающим над несколькими проектами. Они могут сгруппировать в один каталог все файлы, связанные с одним проектом. Каталоги также хорошо подходят для организации совместного доступа к файлам в рабочих группах.

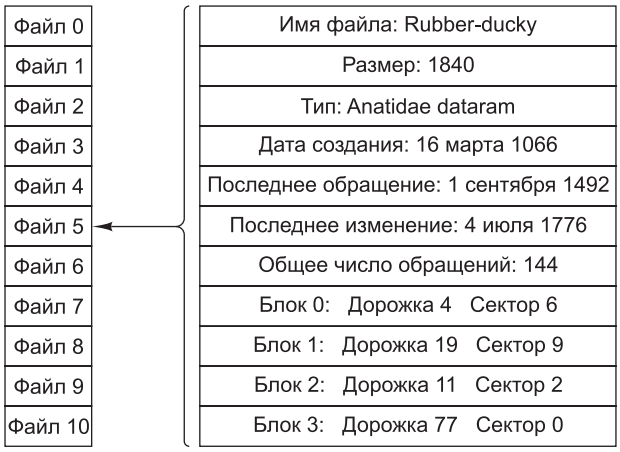


Рис. 6.22. Пользовательский каталог и структура одного из типичных файлов

Виртуальные команды для параллельной работы

Некоторые вычисления удобно производить с помощью двух и более параллельных процессов (то есть будто бы на разных процессорах). Другие вычисления можно поделить на части, которые затем выполняются параллельно. Чтобы несколько процессов могли выполняться параллельно, нужны специальные виртуальные команды. Мы обсудим их в следующих подразделах.

Интерес к параллельной работе обусловлен и некоторыми законами физики. Согласно теории относительности Эйнштейна скорость передачи электрических сигналов не может превышать скорость света, которая равно примерно 1 фут/нс в вакууме, а в медном проводе или оптическом волокне — еще меньше. При разработке компьютеров важно учитывать этот предел. Например, если процессору нужны данные из основной памяти, которые находятся на расстоянии одного фута от него, потребуется, по крайней мере, 1 нс, чтобы запрос дошел до памяти, и еще 1 нс, чтобы ответ вернулся к центральному процессору. Следовательно, чтобы компьютеры могли передавать сигналы быстрее, они (компьютеры) должны быть совершенно крошечными. Альтернативный способ повышения быстродействия компьютера — создание машины с несколькими процессорами. Компьютер, содержащий 1000 процессоров с временем цикла 1 нс каждый, будет иметь такие же возможности, как процессор с временем цикла 0,001 нс, но первый реализовать гораздо проще и дешевле. Параллельные вычисления более подробно рассматриваются в главе 8.

В компьютере с несколькими процессорами каждый из взаимодействующих процессов можно приписать какому-то определенному процессору, чтобы выполнять несколько действий одновременно. Если же в компьютере имеется только один процессор, эффект параллельной работы можно смоделировать. Для этого процессы могут выполняться по очереди, каждый по очень короткому

промежутку времени. Иными словами, процессор будет совместно использоваться несколькими процессами.

На рис. 6.23 показана разница между реальной параллельной работой, когда физически существуют несколько процессоров, и смоделированной параллельной работой, когда физически имеется всего один процессор. Даже если параллелизм моделируется, удобно считать, что каждому процессу приписан собственный виртуальный процессор. При смоделированной параллельной работе возникают те же проблемы, что и при реальной. В обоих случаях отладка создает массу проблем.

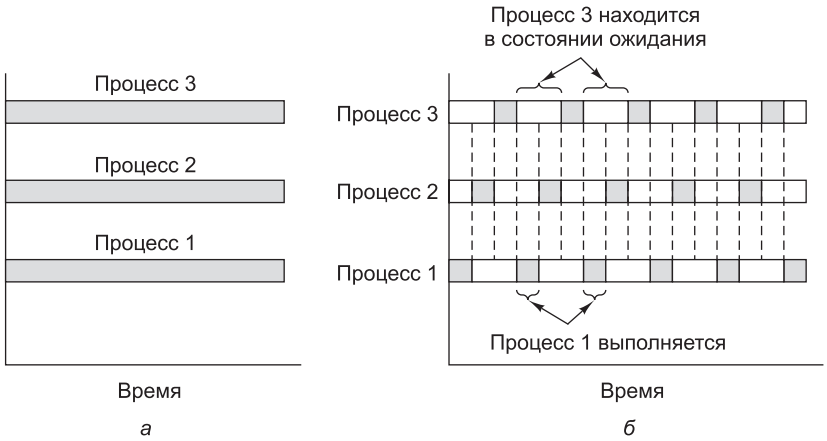


Рис. 6.23. Параллельная работа на нескольких процессорах (*а*); моделирование параллельной работы трех процессов путем переключения единственного процессора с одного процесса на другой (*б*)

Формирование процесса

Исполняемая программа запускается как часть какого-либо процесса. Этот процесс, как и все другие процессы, характеризуется состоянием и адресным пространством, через которое можно получить доступ к программам и данным. Информация о состоянии включает как минимум счетчик команд, слово состояния программы, указатель стека и регистры общего назначения.

Большинство современных операционных систем позволяют формировать и прерывать процессы динамически. Для формирования нового процесса требуется системный вызов. Этот системный вызов может просто создать клон вызывающей программы или позволить исходному процессу задать начальное состояние нового процесса, включая его программу, данные и начальный адрес.

В одних случаях исходный (родительский) процесс может сохранять частичный или даже полный контроль над порожденным (дочерним) процессом. Виртуальные команды позволяют родительскому процессу останавливать и снова запускать, проверять и завершать дочерние процессы. В других случаях исходный процесс никак не контролирует дочерний процесс: после того как новый процесс сформирован, исходный процесс не может его остановить, запустить заново, проверить или завершить. Таким образом, эти два процесса работают независимо друг от друга.

Состояние гонок

Во многих случаях параллельные процессы должны взаимодействовать и синхронизироваться. В этом подразделе мы рассмотрим синхронизацию и некоторые связанные с этим проблемы. Способы разрешения этих проблем будут представлены в следующем разделе.

Рассмотрим два независимых процесса, процесс 1 и процесс 2, которые взаимодействуют через общий буфер в основной памяти. Для простоты будем называть процесс 1 **производителем** (producer), а процесс 2 — **потребителем** (consumer). Производитель генерирует простые числа и помещает их в буфер по одному. Потребитель по одному извлекает их из буфера и печатает.

Эти два процесса работают параллельно с разной скоростью. Если производитель обнаруживает, что буфер заполнен, он переходит в режим ожидания, то есть временно приостанавливает операцию и ожидает сигнала от потребителя. Когда потребитель удаляет число из буфера, он посылает сигнал производителю, чтобы тот возобновил работу. Если потребитель обнаруживает, что буфер пуст, он приостанавливает работу. Когда производитель помещает число в пустой буфер, он посылает соответствующий сигнал потребителю.

В нашем примере для взаимодействия процессов мы задействуем кольцевой буфер. Указатели `in` и `out` используются следующим образом: `in` указывает на следующее свободное слово (куда производитель сможет поместить очередное число), а `out` — на следующее число, которое должен извлечь потребитель. Если `in` = `out`, буфер пуст, как показано на рис. 6.24, *а*. На рис. 6.24, *б* показана ситуация после того, как производитель сгенерировал несколько чисел. На рис. 6.24, *в* изображен буфер после того, как потребитель извлек из него несколько чисел для печати. На рис. 6.24, *г–е* представлены промежуточные этапы работы буфера. Буфер заполняется циклически. Если в буфер отправлено слишком много чисел и он начинает заполняться по второму кругу (снизу), а под адресом `out` есть только одно свободное слово (например, `in` = 52, а `out` = 53), буфер заполнится. Последнее слово не используется; в противном случае не было бы возможности сообщить, что именно значит равенство `in` = `out`, полный буфер или пустой.

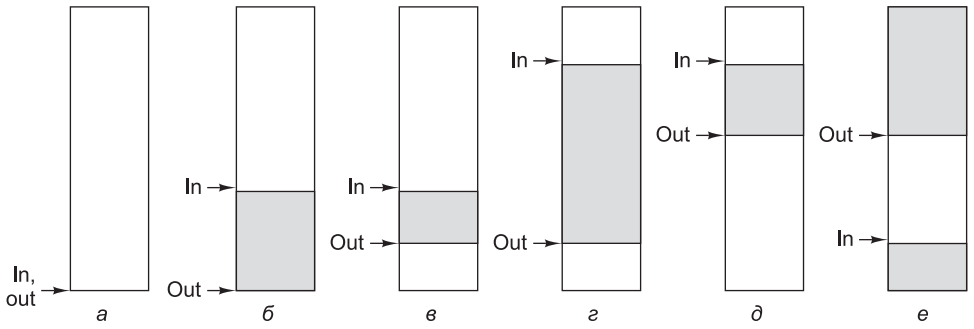


Рис. 6.24. Кольцевой буфер

В листинге 6.1 показано решение задачи с производителем и потребителем на языке Java. Здесь имеются три класса: `m`, `producer` и `consumer`. Класс `m` содержит

некоторые константы, указатели буфера `in` и `out`, и сам буфер, который в нашем примере вмещает 100 простых чисел (от `buffer[0]` до `buffer[99]`).

Листинг 6.1. Параллельная работа в условиях гонок

```
public class m {
  final public static int BUF_SIZE = 100;       // буфер от 0 до 99
  final public static long MAX_PRIME = 100000000000L; // остановиться здесь
  public static int in = 0, out = 0;            // указатели на данные
  public static long buffer[ ] = new long[BUF_SIZE]; // числа хранятся здесь
  public static producer p;                     // имя производителя
  public static consumer c;                     // имя потребителя

  public static void main(String args[ ]){      // основной класс
    p = new producer );                         // создание производителя
    c = new consumer );                         // создание потребителя
    p.start();                                  // запуск производителя
    c.start();                                  // запуск потребителя
  }
// Служебная функция для циклического увеличения in и out
  public static int next(int k) { if (k < BUF_SIZE - 1) return(k + 1);
    else return(0); }
}

class producer extends Thread {                 // класс производителя
  public void run() {                           // код производителя
    long prime = 2;                             // временная переменная

    while (prime < m.MAX__PRIME) {
      prime = next_prime(prime);                // P1
      if (m.next(m.in) == m.out) suspend();     // P2
      m. buffer[m. in] = prime;                 // P3
      m.in = m.next(m.in);                      // P4
      if (m.next(m.out) == m.in) m.c.resume(); // P5
    }
  }

  private long next_prime(long prime){ ... }    // функция, вычисляющая
                                                // следующее число
}

class consumer extends Thread {                 // класс потребителя
  public void run() {                           // код потребителя
    long emirp = 2;                             // временная переменная

    while (emirp < m.MAX_PRIME) {
      if (m.in == m.out) suspend();             // C1
      emirp = m.buffer[m.out];                  // C2
      m.out = m.next(m.out);                    // C3
      if (m.out == m.next(m.next(m.in))) m.p.resume(); // C4
      System.out.println(emirp);                // C5
    }
  }
}
```

Для моделирования параллельных процессов в данном случае используются **программные потоки** (threads). У нас есть классы `producer` и `consumer`, экзем-

пляры которых создаются в переменных p и c соответственно. Каждый из этих классов является производным от базового класса Thread. Метод run этого класса содержит код программного потока. Когда вызывается метод start для объекта, производного от класса Thread, запускается новый поток.

Каждый программный поток похож на процесс. Единственным отличием является то, что все потоки в пределах одной Java-программы находятся в едином адресном пространстве. Это позволяет им разделять один общий буфер. Если в компьютере имеются два и более процессоров, каждый поток может выполняться на собственном процессоре, поэтому в данном случае имеет место реальный параллелизм. Если компьютер содержит только один процессор, потоки разделяются во времени на одном процессоре. Мы будем продолжать называть производителя и потребителя процессами (поскольку нас в данный момент интересуют параллельные процессы), хотя Java поддерживает параллелизм только на уровне программных потоков, а не «настоящих» процессов.

Функция next позволяет увеличивать значения in и out, при этом не нужно каждый раз писать код, чтобы проверять условие окончания цикла. Если параметр в next равен 98 или ниже, то возвращается следующее по порядку целое число. Если параметр равен 99, это значит, что мы достигли конца буфера, поэтому возвращается 0.

Если какой-то из процессов не может продолжаться, желательно иметь способ временно его приостановить. Зная об этой потребности, разработчики первой версии Java включили в класс Thread специальные методы suspend (приостановка) и resume (возобновление работы). Они используются в листинге 6.1.

А теперь рассмотрим сам код производителя и потребителя. Сначала производитель генерирует новое число (P1). Обратите внимание на идентификатор m.MAX_PRIME. Префикс m здесь указывает, что имеется в виду идентификатор MAX_PRIME, определенный в классе m. По той же причине этот префикс нужен для идентификаторов in, out, buffer и next.

Затем (P2) производитель проверяет, не меньше ли in значения out. Если да (например, in = 62 и out = 63), это означает, что буфер заполнен, и производитель вызывает метод suspend. Если буфер не заполнен, туда помещается новое число (P3), и значение in инкрементируется (P4). Если при проверке в точке P5 обнаруживается, что новое значение in на единицу больше, чем out (например, in = 17, out = 16), это означает, что значения in и out перед увеличением in были равны. Тогда производитель делает вывод, что буфер был пуст, потребитель не функционировал (был приостановлен) и продолжает простаивать. Поэтому производитель вызывает метод resume, чтобы заставить потребителя возобновить работу (P5). После этого производитель начинает формировать следующее число.

Программа потребителя имеет сходную структуру. Сначала проверяется, пуст ли буфер (C1). Если пуст, то потребителю ничего не нужно делать, поэтому он отключается. Если буфер не пуст, то потребитель извлекает из него следующее число для печати (C2) и инкрементирует значение out (C3). Если после этого значение out становится на две единицы больше in (C4), это означает, что на предыдущем шаге значение out было на единицу меньше in. Так как условием заполнения буфера как раз и является значение in = out − 1, это означает, что производитель был приостановлен, и потребитель вызывает для производителя метод resume. После этого число выводится на печать (C5), и весь цикл повторяется.

К сожалению, программа содержит ошибку (рис. 6.25). Напомним, что наши два процесса работают асинхронно, то есть с разными скоростями, которые, к тому же, могут меняться. Рассмотрим случай, когда в буфере остается только одно число в элементе 21, и `in` = 22, а `out` = 21 (рис. 6.25, *а*). Производитель в операторе P1 ищет число, а потребитель в операторе C5 печатает число с позиции 20. Потребитель заканчивает печатать число, совершает проверку в операторе C1 и извлекает последнее число из буфера в операторе C2. Затем он инкрементирует значение `out`. В этот момент и `in`, и `out` равны 22. Потребитель печатает число, а затем переходит к оператору C1, в котором он вызывает значения `in` и `out` из памяти, чтобы сравнить их (рис. 6.25, *б*).

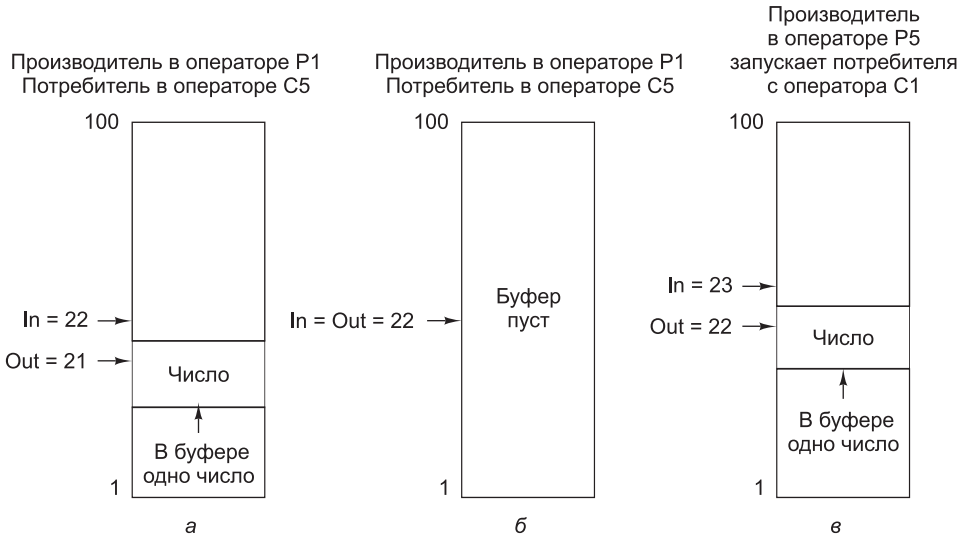


Рис. 6.25. Ситуация, при которой механизм взаимодействия производителя и потребителя не срабатывает

В этот момент, после того как потребитель вызвал значения `in` и `out`, но еще до того как он сравнил их, производитель генерирует следующее число. Он помещает это число в буфер в точке P3 и инкрементирует `in` в точке P4. Теперь `in` = 23, а `out` = 22. В точке P5 производитель обнаруживает, что `in` на единицу больше `out`, а это означает, что в буфере находится одно число. Исходя из этого, производитель делает неверный вывод о том, что потребитель приостановлен, и вызывает для его запуска метод `resume`, как показано на рис. 6.25, *в*. На самом деле потребитель все это время продолжает работать, поэтому вызов метода `resume` оказывается лишним. Затем производитель начинает формировать следующее число.

В этот момент потребитель продолжает работать. Он успевает вызвать значения `in` и `out` из памяти раньше, чем производитель поместил последнее число в буфер. Так как `in` = 22 и `out` = 22, потребитель приостанавливается. К этому моменту производитель генерирует следующее число. Он проверяет указатели и обнаруживает, что `in` = 24, а `out` = 22. Из этого он делает заключение, что в буфере находятся два числа (что соответствует действительности), а потребитель функционирует (что неверно). Производитель продолжает цикл. В конце концов,

он заполняет буфер и приостанавливается. После этого оба процесса остаются приостановленными до скончания веков.

Сложность здесь в том, что между моментом, когда потребитель вызывает `in` и `out`, и моментом, когда он отключается, производитель, обнаружив, что `in` = `out` + 1 и предположив, что потребитель приостановлен (хотя на самом деле он продолжает функционировать), вызывает метод `resume`, чего делать не нужно, поскольку потребитель и так функционирует. Такая ситуация называется **состоянием гонок**, поскольку успех метода зависит от того, кто после инкремента `out` выиграет гонку по проверке `in` и `out`.

Проблема гонок хорошо известна. Она настолько серьезна, что через несколько лет после создания Java компания Sun переработала класс `Thread`, изъяв из него вызовы методов `suspend` и `resume`, поскольку они очень часто приводили к состоянию гонок. Предложенное решение заключается в изменении языка, но поскольку наша цель — операционные системы, мы обсудим другое решение, которое используется во многих операционных системах, в том числе UNIX и Windows 7.

Синхронизация процесса с использованием семафоров

Проблему гонок можно разрешить, по крайней мере, двумя способами. Первый способ — снабдить каждый процесс специальным битом ожидания пробуждения. Этот бит устанавливается при получении сигнала запуска процессом, который продолжает работать. Если процесс приостанавливается при установленном бите, то процесс немедленно перезапускается, а бит сбрасывается. То есть бит ожидания как бы сохраняет сигнал запуска для будущего использования.

Однако этот метод решает проблему гонок только для двух процессов. В общем случае при наличии n процессов он не работает. Конечно, каждому процессу можно приписать $n - 1$ таких битов ожидания пробуждения, но это неудобно.

Эдгар Дейкстра [1968b] предложил другое решение этой проблемы. Где-то в памяти находятся две переменные, которые могут содержать неотрицательные целые переменные, называемые **семафорами**. Операционная система предоставляет два системных вызова, `up` и `down`, которые оперируют семафорами. Вызов `up` прибавляет единицу к значению семафора, а вызов `down` уменьшает значение семафора на единицу.

Если операция `down` совершается для семафора, значение которого больше 0, значение этого семафора уменьшается на единицу, и процесс продолжается. Если же значение семафора равно 0, то операция `down` не может завершиться. Тогда данный процесс приостанавливается до тех пор, пока другой процесс не выполнит для этого семафора операцию `up`. Как правило, приостановленные процессы собираются в одну очередь, что упрощает задачу их отслеживания.

Команда `up` проверяет, не равно ли 0 значение семафора. Если равно, а другой процесс приостановлен, значение семафора увеличивается на единицу. После этого приостановленный процесс завершит операцию `down`, установив в 0 значение семафора. В этом случае оба процесса могут продолжать работу. Если значение семафора не равно 0, команда `up` просто увеличивает его на единицу. Семафор позволяет сохранять сигналы пробуждения, так что они не пропадут зря. У семафорных команд есть одно важное свойство: если один из процессов

начал выполнять команду для семафора, то другой процесс не сможет получить доступ к этому семафору до тех пор, пока первый процесс не завершит выполнение команды или не будет приостановлен, пытаясь выполнить команду down при нулевом значении семафора. В табл. 6.2 представлены важные свойства системных вызовов up и down.

Таблица 6.2. Результаты операций с семафором

Команда	Семафор = 0	Семафор > 0
up	Семафор = семафор + 1 (если другой процесс пытается теперь выполнить операцию down для этого семафора, он сможет это сделать и продолжить свою работу)	Семафор = семафор +1
down	Процесс приостанавливается до тех пор, пока другой процесс не выполнит операцию up для этого семафора	Семафор = семафор –1

Как уже отмечалось, проблема гонок может быть решена средствами языка Java, но мы сейчас обсуждаем операционные системы, следовательно, нам нужно каким-либо образом реализовать механизм семафоров на языке Java, поскольку стандартного класса для них нет. Однако мы сможем это сделать, предположив, что два метода, up и down, совершающие соответственно системные вызовы up и down, уже написаны и используют в качестве их параметров обычные целые числа.

В листинге 6.2 показано, как можно устранить состояние гонок с помощью семафоров. В класс m добавляются два семафора: семафор available, изначально равный 100 (это размер буфера), и семафор filled, изначально равный 0. Производитель начинает работу с оператора P1, а потребитель — с оператора C1. Вызов down для семафора filled сразу же приостанавливает работу потребителя. Когда производитель находит первое число, он вызывает метод down с переменной available в качестве параметра, присваивая ей значение 99. В операторе P5 производитель вызывает метод up с параметром filled, присваивая переменной filled значение 1. Это действие освобождает потребителя, который теперь может завершить вызов метода down. После этого filled принимает значение 0, и оба процесса продолжают работу.

Листинг 6.2. Параллельная работа с использованием семафоров

```
public class m {
  final public static int BUF_SIZE = 100;            // буфер от 0 до 99
  final public static long MAX_PRIME=100000000000L;  // остановиться здесь
  public static int in = 0, out = 0;                 // указатели на данные
  public static long buffer[ ] = new long[BUF_SIZE]; // здесь хранятся числа
  public static producer p;                          // имя производителя
  public static consumer c;                          // имя потребителя
  public static int filled = 0, available = 100;     // семафоры

  public static void main(String args[ ]) {       // основной класс
    p = new producer();                           // создание производителя
    c = new consumer();                           // создание потребителя
    p.start();                                    // запуск производителя
    c.start();                                    // запуск потребителя
  }
```

```
  // Это прикладная функция для циклического увеличения in и out
  public static int next(int k) {if (k < BUF_SIZE - 1) return(k+1);
    else return(0);}
}

class producer extends Thread {                    // класс производителя
  native void up(int s); native void down(int s); // методы для семафоров
  public void run() {                              // код производителя
    long prime = 2;                                // временная переменная

    while (prime < m.MAX_PRIME) {
      prime = next_prime(prime);                   // P1
      down(m.available);                           // P2
      m.buffer[m.in] = prime;                      // P3
      m.in = m.next(m.in);                         // P4
      up(m.filled);                                // P5
    }
  }
  // Функция, которая вычисляет следующее число
  private long next_prime(long prime){ ...}
}

class consumer extends Thread {                    // класс потребителя
  native void up(int s); native void down(int s); // методы для семафоров
  public void run() {                              // код потребителя
    long emirp = 2;                                // временная переменная

    while (emirp < m.MAX_PRIME) {
      down(m.filled);                              // C1
      emirp = m.buffer[m.out];                     // C2
      m.out = m.next(m.out);                       // C3
      up(m. available);                            // C4
      System.out.println(emirp);                   // C5
    }
  }
}
```

Давайте еще раз вернемся к состоянию гонок. Пусть в определенный момент `in` = 22, а `out` = 21, выполнение производителя находится в точке P1, а потребителя — в точке C5. Потребитель завершает действие и переходит в точку C1, в которой вызывается метод `down` для семафора `filled`, до вызова имевшего значение 1, а после вызова получившего значение 0. Затем потребитель извлекает последнее число из буфера и вызывает метод `up` для семафора `available`, после чего `available` принимает значение 100. Потребитель печатает это число и переходит в точку C1. Как раз перед тем, как потребитель собирается вызвать метод `down`, производитель формирует следующее число и быстро выполняет операторы P2, P3 и P4.

В этот момент `filled` = 0. Производитель собирается вызывать для этого семафора метод `up`, а потребитель — метод `down`. Если потребитель сделает вызов первым, он будет приостановлен до тех пор, пока производитель его не освободит (вызвав метод `up`). Если же первым вызов сделает производитель, то семафор получит значение 1, и потребитель вообще не будет приостановлен. В обоих случаях сигнал запуска не пропадет. Именно для этого мы и ввели в программу семафоры.

Операции с семафорами неделимы. Если операция с семафором уже началась, то никакой другой процесс не может использовать этот семафор до тех пор, пока первый процесс не завершит операцию или пока он не будет приостановлен. Более того, при наличии семафоров сигналы запуска не пропадают. А вот операторы `if` в листинге 6.1 делимы. Между проверкой условия и выполнением нужного действия другой процесс может послать сигнал запуска.

В сущности, проблема синхронизации была решена путем введения неделимых системных вызовов `up` и `down`. Чтобы эти операции были неделимыми, операционная система должна запретить двум и более процессам использовать один и тот же семафор одновременно. То есть если делается системный вызов `up` или `down`, никакой пользовательский код не может быть запущен, пока вызов не завершится. Как правило, в однопроцессорных системах для этого во время выполнения операций с семафорами вводится запрет на прерывания. В мультипроцессорных системах этот прием не проходит.

Синхронизация с использованием семафоров работает для произвольного количества процессов. Несколько процессов могут приостановиться, пытаясь выполнить системный вызов `down` для одного и того же семафора. Когда какой-нибудь другой процесс выполнит системный вызов `up` для этого же семафора, один из приостановленных процессов может завершить вызов `down` и продолжить работу. В этом случае семафор сохранит значение 0, и другие процессы останутся приостановленными.

Поясним ситуацию на другом примере. Представьте себе 20 волейбольных команд, играющих 10 партий (процессов). Каждая игра происходит на отдельном поле. Имеется большая корзина (семафор) для волейбольных мячей. К сожалению, мячей только 7. В каждый момент в корзине находится от 0 до 7 мячей (семафор принимает значение от 0 до 7). Помещение мяча в корзину — это операция `up`, поскольку она увеличивает значение семафора. Извлечение мяча из корзины — это операция `down`, поскольку она уменьшает значение семафора.

В самом начале один игрок от каждого поля посылается к корзине за мячом. Семерым из них удается получить мяч (завершить операцию `down`); оставшиеся трое вынуждены ждать. Их игры временно приостановлены. В конце концов, одна из партий заканчивается, и мяч возвращается в корзину (выполняется операция `up`). Эта операция позволяет одному из трех оставшихся игроков получить мяч (закончить незавершенную операцию `down`) и продолжить игру. Оставшиеся две партии остаются приостановленными до тех пор, пока еще два мяча не возвратятся в корзину. Когда эти два мяча попадут в корзину (то есть выполнятся еще две операции `up`), можно будет продолжить последние две партии.

Примеры операционных систем

В этом разделе мы продолжим обсуждать процессоры Core i7 и OMAP4430 и рассмотрим операционные системы, которые используются с этими процессорами. Для Core i7 мы возьмем Windows, а для OMAP4430 — UNIX. Начнем мы с UNIX, поскольку эта система гораздо проще и во многих отношениях элегантнее Windows. А поскольку операционная система UNIX была спроектирована

раньше и оказала существенное влияние на Windows 7, такой порядок изложения вполне логичен.

Знакомство с операционными системами UNIX и Windows XP

В этом подразделе мы дадим краткий обзор двух операционных систем (UNIX и Windows 7). При этом особое внимание будет уделено истории, структуре и системным вызовам.

UNIX

Операционная система UNIX была разработана в компании Bell Labs в начале 70-х годов. Первая версия была написана Кеном Томпсоном (Ken Thompson) на ассемблере для мини-компьютера PDP-7. Затем появилась вторая версия для компьютера PDP-11, уже на языке C. Ее автором был Деннис Ритчи (Dennis Ritchie). В 1974 году Ритчи и Томпсон опубликовали очень важную работу о системе UNIX [Ritchie and Thompson, 1974]. За эту работу они были награждены престижной премией Тьюринга Ассоциации вычислительной техники [Ritchie, 1984; Thompson, 1984]. После публикации многие университеты попросили у Bell Labs копию UNIX. Поскольку материнская компания Bell Labs, AT&T, была в тот момент регулируемой монополией и ей не разрешалось участвовать в компьютерном бизнесе, университеты смогли приобрести лицензию на UNIX за небольшую плату.

Машины PDP-11 использовались практически во всех компьютерных научных отделах университетов, но операционные системы, которые пришли туда вместе с PDP-11, не нравились ни профессорам, ни студентам. Эту нишу быстро заполнила операционная система UNIX, которая была снабжена исходными текстами, поэтому желающие могли дорабатывать ее до бесконечности.

Одним из первых университетов, купивших UNIX, был Калифорнийский университет в Беркли. Поскольку в наличии имелись все исходные коды, в Беркли сумели существенно преобразовать UNIX. Среди изменений было перенесение этой системы на мини-компьютер VAX, создание виртуальной памяти со страничной организацией, расширение имен файлов с 14 до 255 символов, а также поддержка сетевого протокола TCP/IP, который сейчас широко используется в Интернете (во многом благодаря тому факту, что он был задействован в системе Berkeley UNIX).

Пока в Беркли шла вся эта работа, компания AT&T самостоятельно продолжала совершенствовать UNIX, в результате в 1982 году появилась операционная система System III, а в 1984 — System V. В конце 80-х годов широко использовались две разные и совершенно несовместимые версии UNIX: Berkeley UNIX и System V. Такое положение вещей вместе с отсутствием стандартов на форматы программ в двоичном коде в значительной степени препятствовало коммерческому успеху UNIX. Поставщики программного обеспечения не могли писать программы для UNIX, поскольку не было никакой гарантии, что эти программы смогут работать на любой версии UNIX (в этом отношении у MS-DOS было значительное преимущество). После долгих споров организация по стандартам при институте IEEE (Institute of Electrical and Electronic Engineers — Институт инженеров по электро-

технике и электронике) выпустила стандарт **POSIX** (Portable Operating System Interface — интерфейс переносимых операционных систем), известный также как стандарт IEEE P1003. Позднее этот стандарт стал международным.

Стандарт POSIX состоит из нескольких частей, каждая из которых посвящена отдельной области UNIX. Первая часть, P1003.1, определяет системные вызовы; вторая часть, P1003.2, — основные прикладные программы и т. д. Стандарт P1003.1 определяет около 60 системных вызовов, которые должны поддерживаться всеми системами. Это вызовы для чтения и записи файлов, создания новых процессов и т. д. Сейчас практически все системы UNIX поддерживают системные вызовы, перечисленные в P1003.1, однако многие системы поддерживают и дополнительные системные вызовы, в частности те, что были реализованы в System V и в Berkeley UNIX. Обычно к набору POSIX добавляются до 200 системных вызовов.

В 1987 году один из авторов этой книги (Таненбаум) выпустил исходный код для одной из версий UNIX, названной MINIX и предназначенной для установки в университетских компьютерных системах [Tanenbaum, 1987]. Одним из студентов, познакомившихся с MINIX в университете Хельсинки и установивших эту систему на домашнем компьютере, оказался Линус Торвальдс (Linus Torvalds). Изучив MINIX в деталях, Линус решил написать собственную версию операционной системы, которая получила имя Linux и стала со временем весьма популярной.

Многие операционные системы, работающие сегодня на платформах ARM, основаны на Linux. Так как и MINIX, и Linux соответствуют стандарту POSIX, практически все, сказанное о UNIX в этой главе, также применимо и к Linux (если в тексте не указано обратное).

В табл. 6.3 перечислены категории системных вызовов Solaris. Системные вызовы управления файлами и каталогами — это самая важная и объемная категория. Система Linux в основном совместима с POSIX P1003.1, хотя разработчики в нескольких местах отклонились от спецификации. Впрочем, построить и запустить POSIX-совместимую программу в Linux обычно бывает несложно.

Таблица 6.3. Системные вызовы UNIX

Категория	Примеры
Управление файлами	Открытие, чтение, запись, закрытие и блокировка файлов
Управление каталогами	Создание и удаление каталогов; перемещение файлов в каталоги
Управление процессами	Порождение, завершение и трассировка процессов, передача сигналов
Управление памятью	Разделение общей памяти между процессами, защита страниц
Получение и установка параметров	Идентификация пользователя, группы, процесса; установка приоритетов
Дата и время	Установка времени доступа к файлам; использование временны́х интервалов; рабочий профиль программы
Работа в сети	Установка и выделение соединений; отправка и получение сообщений
Прочее	Учет использования ресурсов; манипуляция дисковыми квотами; перезагрузка системы

Сфера использования сетей в большей степени относится к Berkeley UNIX, а не к System V. Именно в Беркли было введено понятие **сокета** как конечной точки сетевого соединения. Моделью для этой концепции стали 4-проводные настенные розетки для телефонов. Процесс в системе UNIX может создать сокет, подключиться к нему и установить соединение с сокетом на удаленном компьютере. Через это соединение можно затем пересылать данные в обоих направлениях, обычно по протоколу TCP/IP. Так как сетевые технологии в системе UNIX применялись десятилетиями, они стали очень зрелыми и стабильными, поэтому многие серверы в Интернете используют именно UNIX.

Существует много разных версий UNIX и каждая из них чем-то отличается от всех остальных, поэтому структуру этой операционной системы описать трудно. Тем не менее схема, изображенная на рис. 6.26, применима к большинству из них. Внизу показаны драйверы устройств, которые изолируют файловую систему от аппаратного обеспечения. Изначально каждый драйвер устройства писался без учета всех остальных и представлял собой независимую единицу. Это привело к многочисленным дублированиям, поскольку многие драйверы должны поддерживать управление потоками, исправление ошибок, приоритеты, отделение данных от команд и т. д. По этой причине Деннис Ритчи для использования принципа модульности при написании драйверов придумал структуру под названием **поток ввода-вывода** (stream). При наличии потока ввода-вывода можно установить двухстороннее соединение между пользовательским процессом и устройством и вставить между ними один или несколько модулей. Пользовательский процесс передает данные в поток ввода-вывода, затем эти данные обрабатываются или просто передаются дальше каждым модулем до тех пор, пока не дойдут до устройства. При передаче данных от устройства процессу все происходит аналогично.

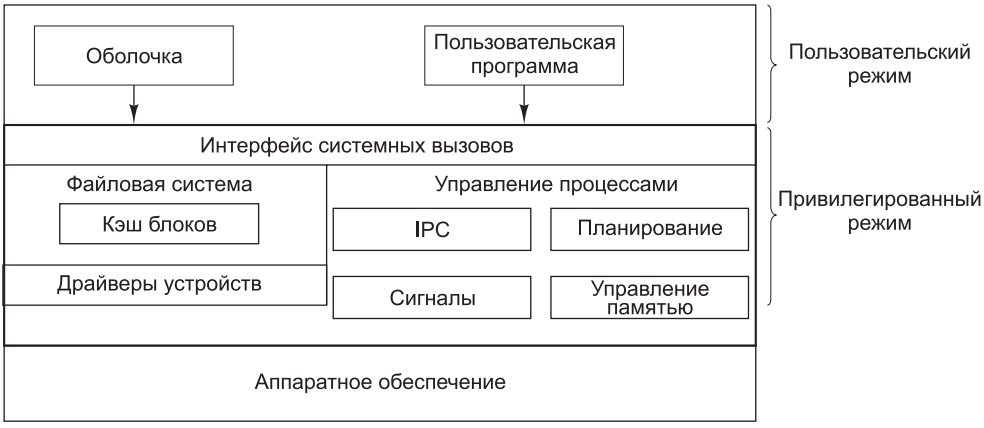


Рис. 6.26. Структура типичной системы UNIX

Над драйверами устройств находится файловая система. Она управляет файлами, каталогами, размещением дисковых блоков, защитой и выполняет многие другие функции. В системе файлов имеется так называемый **кэш блоков** (block cache), предназначенный для хранения недавно считанных с диска блоков на случай, если они понадобятся еще раз. Некоторые файловые системы исполь-

зовались многие годы. Среди них можно назвать быструю файловую систему Berkeley [McKusick et al., 1984] и файловые системы с журнальной структурой [Rosenblum and Ousterhout, 1991; Seltzer et al., 1993].

Еще одна часть ядра системы UNIX — механизм управления процессами. Он выполняет различные функции, в том числе поддерживает взаимодействие между процессами (InterProcess Communication, IPC) и их синхронизацию, что позволяет избежать состояния гонок. Код управления процессами также занимается планированием процессов на основе их приоритетов. Кроме того, он обрабатывает сигналы, которые представляют собой особую (асинхронную) форму программных прерываний. Наконец, он же управляет памятью. Большинство систем UNIX поддерживают виртуальную память с подкачкой страниц по требованию, иногда с некоторыми дополнительными особенностями (например, несколько процессов могут совместно использовать общие области адресного пространства).

UNIX изначально задумывалась как весьма компактная система; компактность была призвана обеспечить повышение надежности и производительности. Первые версии UNIX были полностью текстовыми и ориентировались на терминалы, которые могли отображать 24 или 25 строк по 80 ASCII-символов. Пользовательским интерфейсом управляла программа, которая называлась **оболочкой** и предоставляла интерфейс командной строки. Поскольку оболочка не является частью ядра, в UNIX легко добавлять новые оболочки, и с течением времени было разработано несколько чрезвычайно сложных оболочек.

Позднее, когда появились графические терминалы, в Массачусетском технологическом институте для UNIX была разработана оконная система, получившая название **X Windows**. Еще позже поверх X Windows был установлен полнофункциональный **графический пользовательский интерфейс** (Graphical User Interface, **GUI**), названный **Motif**. Поскольку ядро должно было оставаться компактным, практически весь код для X Windows и Motif работает вне ядра в пользовательском режиме.

Windows 7

Первая машина IBM PC, выпущенная в 1981 году, была оснащена 16-разрядной операционной системой реального режима, ориентированной на индивидуального пользователя и имевшей интерфейс командной строки. Она называлась MS-DOS 1.0. Эта операционная система представляла собой 8-килобайтный код, находящийся в памяти. Через два года появилась более мощная 24-килобайтная система MS-DOS 2.0. Она содержала процессор командной строки (оболочку), и многие ее функции были заимствованы у системы UNIX. В 1984 году компания IBM выпустила машину PC/AT с операционной системой MS DOS-3.0, объем которой к тому моменту составлял уже 36 Кбайт. С годами у системы MS-DOS появлялись все новые и новые функции, но она при этом оставалась системой командной строки.

Вдохновленная успехом Apple Macintosh, компания Microsoft решила создать графический пользовательский интерфейс, который она назвала **Windows**. Первые три версии Windows, включая Windows 3.x, были не «настоящими» операционными системами, а графическими пользовательскими интерфейсами на базе MS-DOS. Все программы работали в одном и том же адресном пространстве, и ошибка в любой из них могла привести к «зависанию» всей системы.

В 1995 году появилась система Windows 95, но она не знаменовала собой отказ от MS-DOS, просто на смену прежней версии MS-DOS пришла новая — MS-DOS 7.0. Windows 95 и MS-DOS 7.0 в совокупности имели характерные особенности развитой операционной системы, в том числе поддерживали виртуальную память, управление процессами и мультипрограммирование. Однако операционная система Windows 95 не была полностью 32-разрядной программой. Она содержала большие фрагменты старого 16-разрядного кода (с небольшими вкраплениями 32-разрядного) и по-прежнему использовала файловую систему MS-DOS со всеми ее недостатками. Единственными значительными изменениями в файловой системе стали увеличение длины имена файлов (ранее в MS-DOS длина имен файлов не должна была превышать 8 + 3 символов) и снятие прежнего ограничения (равного 65 536) на количество дисковых блоков.

Даже в системе Windows 98, вышедшей в 1998 году, 16-разрядный код MS-DOS (на этот раз версии 7.1) все еще присутствовал. Система Windows 98 не слишком отличалась от Windows 95, хотя часть функций перекочевали от MS-DOS к Windows, а дисковый формат, подходящий для дисков большего размера, стал стандартным. Основным отличием был пользовательский интерфейс, который объединил рабочий стол, Интернет и даже в какой-то степени телевидение, сделав систему более автономной. Именно это и привлекло внимание министерства юстиции США, которое обвинило Microsoft в нарушении антимонопольного законодательства. Через некоторое время в свет вышла несколько усовершенствованная редакция Windows 98 под именем Windows Millenium Edition (ME), но просуществовала она недолго.

Во время всех этих событий компания Microsoft разрабатывала совершенно новую 32-разрядную операционную систему. Эта новая система называлась **Windows New Technology** (новая технология Windows), или сокращенно **Windows NT**[1]. Изначально предполагалось, что она заменит все операционные системы, предназначенные для компьютеров на базе процессоров Intel (а также на базе PowerPC), онако ее распространение шло очень медленно и позднее была переориентирована на более мощные компьютеры. В этом сегменте Windows NT нашла применение на больших серверах. Вторая версия Windows NT, названная Windows 2000, имела гораздо больший успех, в том числе и в сегменте домашних компьютеров. Последовательницей Windows 2000 стала система Windows XP, однако отличия не столь значительны (в основном улучшенная обратная совместимость и ряд дополнительных функций). В 2007 году была выпущена следующая система из семейства Windows — Windows Vista. В ней были реализованы многочисленные графические усовершенствования, а также добавлены новые приложения (например, Медиацентр). Пользователи не торопились переходить на Vista из-за низкой производительности и высоких требований

[1] Работы над операционной системой, получившей впоследствии название Windows NT, начались после ряда неудач с 16-разрядной версией OS/2 в рамках совместного проекта IBM и Microsoft по созданию новой 32-разрядной операционной системы OS/2. Этот проект, ориентированный на микропроцессор i386, начался в 1989 году, однако уже в следующем году пути этих компаний разошлись, и Microsoft, продолжившая работу над OS/2 v.3.0, дала ей имя Windows NT, желая тем самым, во-первых, дистанцироваться от IBM, а во-вторых, подчеркнуть, что использует единый графический интерфейс со всеми своими популярными одноименными оболочками. — *Примеч. науч. ред.*

к ресурсам. Всего через два года была выпущена система Windows 7, которая во всех отношениях представляет собой оптимизированную версию Windows Vista. Windows 7 намного лучше работает на старом оборудовании и требует намного меньше аппаратных ресурсов.

Windows 7 продается в шести разных версиях. Три версии предназначены для домашних пользователей в разных странах, две — для пользователей в сфере бизнеса, и одна объединяет в себе возможности всех версий. Версии почти идентичны; они отличаются только направленностью, расширенными функциями и оптимизациями. Мы ограничимся рассмотрением основных функций, не отвлекаясь на обсуждение различий между версиями.

Прежде чем переходить к интерфейсу, который Windows 7 предоставляет программисту, давайте очень кратко рассмотрим внутреннюю структуру системы, изображенную на рис. 6.27. Она состоит из ряда модулей, распределенных по уровням и совместно реализующих операционную систему. Каждый модуль выполняет определенную функцию и имеет определенный интерфейс с другими модулями. Практически все модули написаны на языке C, хотя часть графического интерфейса написана на C++, а кое-что из самых нижних уровней — на ассемблере.

Рис. 6.27. Структура Windows 7

В самом низу расположен **уровень аппаратных абстракций**. Он должен предоставлять операционной системе некие абстрактные устройства, лишенные всех недостатков и странностей, на которые так богаты реальные устройства. К моделируемым устройствам относятся кэш-память, расположенная вне микросхемы, тактовые генераторы, шины ввода-вывода, контроллеры прерываний, контроллеры прямого доступа к памяти. Если эти устройства предоставить операционной системе в идеализированном виде, это упростит перенос Windows 7 на другие аппаратные платформы, поскольку большую часть изменений потребуется сделать только в одном месте.

Над уровнем аппаратных абстракций код делится на две крупные подсистемы: **исполнительный уровень NTOS** и **драйверы Windows**, включающие код файло-

вых систем, поддержки сети и графики. Над ними находится уровень ядра. Весь этот код выполняется в защищенном привилегированном режиме.

Исполнительный уровень управляет важнейшими абстракциями Windows 7: программными потоками, процессами, виртуальной памятью, объектами ядра и конфигурациями. Также здесь находятся подсистемы управления локальными вызовами процедур, кэшированием файлов, вводом-выводом и безопасностью.

За пределами ядра находятся пользовательские программы и системная библиотека, используемая для взаимодействия с операционной системой. В отличие от UNIX компания Microsoft не поощряет прямое использование системных вызовов пользовательскими программами. Для обеспечения стандартизации между разными версиями Windows (например, XP, Vista и Windows 7) был определен набор функций, известный как **Win32 API** (Application Programming Interface — **прикладной программный интерфейс**). Это библиотечные функции, которые выполняют определенные действия либо в системе путем системных вызовов, либо в некоторых случаях непосредственно в библиотечной процедуре пользовательского пространства. Хотя с момента определения Win32 было добавлено много новых библиотечных функций Windows 7, базовые функции остались неизменными; именно на них мы сосредоточимся. Позднее система Windows была портирована для 64-разрядных машин, а компания Microsoft изменила название набора с Win32 на Win64, однако для наших целей изучения 32-разрядной версии вполне достаточно.

Философия Win32 API полностью отличается от философии UNIX. В UNIX все системные вызовы общеизвестны и формируют минимальный интерфейс: удаление хотя бы одного из них изменит функционирование операционной системы. Подсистема Win32 предоставляет избыточный интерфейс, в котором одно и то же действие часто можно выполнить тремя или четырьмя разными способами. Кроме того, Win32 включает в себя много функций, которые не являются системными вызовами (например, копирование целого файла).

Многие вызовы Win32 API создают объекты ядра того или иного типа (файлы, процессы, программные потоки, каналы и т. п.). Каждый вызов, ведущий к созданию объекта ядра, возвращает вызывающей программе результат, который называется **идентификатором** (описателем) (handle). Этот описатель впоследствии может применяться для выполнения операций с объектом. Для каждого процесса существует свой описатель. Он не может непосредственно передаваться другому процессу и использоваться этим процессом (дескрипторы файлов в UNIX тоже нельзя передавать другим процессам). Однако при определенных обстоятельствах можно продублировать описатель, передать его другим процессам и разрешить им доступ к объектам, которые принадлежат другим процессам. Каждый объект имеет связанный с ним дескриптор безопасности, который определяет, кому разрешено, а кому запрещено совершать те или иные операции с объектом.

Операционную систему Windows 7 иногда называют объектно-ориентированной, поскольку оперировать объектами ядра можно только по их описателям путем вызова методов (функций API). С другой стороны, она не поддерживает такие основные свойства объектно-ориентированной системы, как наследование и полиморфизм.

Примеры виртуальной памяти

В этом подразделе мы поговорим о виртуальной памяти в UNIX и Windows 7. С точки зрения программиста они во многом сходны.

Виртуальная память UNIX

Модель памяти UNIX довольно проста. Каждый процесс имеет три сегмента: код, данные и стек, как показано на рис. 6.28. В машине с линейным адресным пространством код обычно располагается в нижней части памяти, а за ним следуют данные. Стек помещается в верхней части памяти. Размер кода фиксирован, а данные и стек могут увеличиваться или уменьшаться (в разных направлениях). Такую модель легко реализовать практически на любой машине. В частности, она используется вариантами Linux, работающими на процессорах OMAP4430.

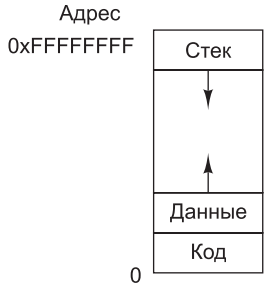


Рис. 6.28. Адресное пространство одного процесса в UNIX

Более того, если машина поддерживает страничную память, то все адресное пространство может быть разбито на страницы совершенно прозрачно для пользовательских программ. Им будет известно только то, что размер программы может превышать размер физической памяти машины. Те системы UNIX, которые не поддерживают страничную организацию памяти, обычно подкачивают целые процессы между памятью и диском, чтобы параллельно могло выполняться произвольное число процессов.

Данное ранее описание системы UNIX (виртуальная память с подкачкой страниц по требованию) в целом соответствует версии Berkeley UNIX, однако версии System V (и Linux) имеют некоторые особенности, позволяющие пользователям управлять виртуальной памятью. Самой важной является способность процесса отображать файл или часть файла на часть адресного пространства процесса. Например, если файл размером 12 Кбайт отображается на виртуальный адрес 144K, то в ячейке с адресом 144K будет находиться первое слово этого файла. Таким образом, можно выполнять ввод-вывод файла без системных вызовов. Поскольку размер некоторых файлов превышает размер виртуального адресного пространства, можно отображать не весь файл, а только его часть. Для отображения сначала нужно открыть файл и получить дескриптор файла `fd` (file descriptor). Дескриптор используется для идентификации отображаемого файла. Затем процесс совершает вызов

```
paddr = mmap(virtual_address, length, protection, flags, fd, file_offset)
```

Этот вызов отображает `length` байт, начиная со смещения `file_offset` в файле, на виртуальное адресное пространство, начиная с адреса `virtual_address`.

Параметр `flags` требует, чтобы система выбрала виртуальный адрес, который затем возвращается в параметре `paddr`. Отображаемая область должна быть выровнена в границах страницы и содержать целое число страниц. Параметр `protection` определяет уровень защиты, включая возможность чтения, записи и выполнения (в любой комбинации). Отображение можно в дальнейшем удалить с помощью команды `unmap`.

Несколько процессов могут одновременно выполнять отображение одного и того же файла. Есть два варианта разделения. В первом варианте общими являются все страницы, поэтому записи, сделанные одним процессом, доступны всем другим процессам. Эта возможность обеспечивает высокоскоростное взаимодействие между процессами. Во втором варианте страницы остаются общими для всех процессов до тех пор, пока ни один из процессов их не меняет. Как только какой-нибудь процесс попытается произвести запись в страницу, он получит ошибку защиты, в результате которой операционная система предоставит ему собственную копию этой страницы для записи. Такая схема, которая называется **копированием при записи** (copy on write), используется в том случае, когда для каждого из нескольких процессов нужно создать иллюзию, что только он выполняет отображение файла. В этой модели совместный доступ является оптимизацией, а не частью семантики.

Виртуальная память Windows 7

В Windows 7 каждый пользовательский процесс имеет собственное виртуальное адресное пространство. В 32-разрядной версии Windows 7 длина виртуального адреса составляет 32 бита, поэтому у каждого процесса есть 4 Гбайт виртуального адресного пространства. Нижние 2 Гбайт предназначены для кода и данных процесса; верхние 2 Гбайт обеспечивают доступ (ограниченный) к памяти ядра. Исключение составляют серверные версии Windows 7, в которых разделение памяти может быть другим (3 Гбайт пользователю и 1 Гбайт ядру). Виртуальное адресное пространство с подкачкой страниц по требованию содержит страницы фиксированного размера (4 Кбайт для Core i7). Адресное пространство 64-разрядной версии Windows 7 имеет аналогичную структуру, но пространство кода и данных в нем находится в младших 8 терабайтах виртуального адресного пространства.

Каждая виртуальная страница может находиться в одном из трех состояний: **свободна** (free), **зарезервирована** (reserved) или **выделена** (committed). Свободная страница в текущий момент не используется, и обращение к ней вызывает ошибку отсутствия страницы. Когда процесс начинается, все его страницы находятся в свободном состоянии до тех пор, пока программа и начальные данные не будут отображены на свое адресное пространство. Если код или данные отображены в страницу, то такая страница является выделенной. Обращение к выделенной странице будет успешным, если страница находится в основной памяти. Если страница отсутствует в основной памяти, произойдет ошибка, и операционной системе придется вызывать нужную страницу с диска. Виртуальная страница может находиться и в зарезервированном состоянии. Это значит, что страница остается недоступной для отображения до тех пор, пока резервирование не будет отменено. Помимо атрибутов состояния страницы, имеются и другие атрибуты (например, указывающие на возможность чтения, записи и выполнения). Верхние 64 Кбайт и нижние 64 Кбайт памяти всегда свободны,

чтобы можно было отыскивать ошибки указателей (неинициализированные указатели часто равны 0 или –1).

Каждая выделенная страница имеет теневую страницу на диске, где она хранится, когда ее нет в основной памяти. Свободные и зарезервированные страницы не имеют теневых страниц, поэтому обращения к ним вызывают ошибки отсутствия страницы (система не может вызвать страницу с диска, если этой страницы нет на диске). Теневые страницы на диске сгруппированы в один или несколько страничных файлов. Операционная система следит, на какую часть какого страничного файла отображается каждая виртуальная страница. Файлы с текстами программ имеют теневые страницы; для страниц данных используются специальные страничные файлы.

Windows 7, как и System V, позволяет отображать файлы прямо на области виртуального адресного пространства. Если файл отображен на адресное пространство, его можно считывать или записывать путем обычных обращений к памяти.

Отображаемые на память файлы реализуются так же, как другие выделенные страницы, только теневые страницы могут находиться в файле на диске, а не в страничном файле. В результате, когда файл отображается, версия в памяти может не совпадать с версией на диске (из-за последних записей в виртуальное адресное пространство). Однако когда отображение файла удаляется, версия на диске обновляется.

Windows 7 позволяет двум и более процессам одновременно отобразить один и тот же файл — возможно, на разные виртуальные адреса. Путем считывания слов из памяти и записи слов в память процессы могут взаимодействовать друг с другом и передавать данные в обоих направлениях с достаточно высокой скоростью, поскольку никакого копирования не требуется. Разные процессы могут обладать разными разрешениями на доступ. Все процессы, работающие с отображенным файлом, разделяют одни и те же страницы, поэтому изменения, произведенные одним из процессов, видны всем остальным процессам, даже если файл на диске еще не обновлен.

Win32 API содержит ряд функций, которые позволяют процессу непосредственно управлять виртуальной памятью. Самые важные из этих функций перечислены в табл. 6.4. Все они выполняются в области, состоящей либо из одной страницы, либо из двух или более страниц, последовательно расположенных в виртуальном адресном пространстве.

Таблица 6.4. Основные функции API для управления виртуальной памятью в Windows XP

Функция API	Описание
VirtualAlloc	Резервирование или выделение области
VirtualFree	Освобождение области или отмена выделения
VirtualProtect	Изменение варианта защиты (чтение/запись/выполнение)
VirtualQuery	Запрос о состоянии области памяти
VirtualLock	Запрещение разбиения памяти на страницы (область памяти становится резидентной)
VirtualUnlock	Снятие запрета относительно разбиения на страницы

Функция API	Описание
CreateFileMapping	Создание объекта отображения файла и назначение (не всегда) ему имени
MapViewOfFile	Отображение файла или части файла на адресное пространство
UnmapViewOfFile	Удаление отображенного файла из адресного пространства
OpenFileMapping	Открытие ранее созданного объекта отображения файла

Первые четыре функции очевидны. Следующие две функции позволяют процессу делать резидентной область памяти размером до 30 страниц и отменять это действие. Это качество может понадобиться программам, работающим в режиме реального времени. Операционная система устанавливает определенный предел, чтобы процессы не становились слишком «прожорливыми». В системе Windows 7 также имеются функции API (они не указаны в табл. 6.4), позволяющие процессу получать доступ к виртуальной памяти другого процесса, который находится под его контролем (то есть для которого он располагает описателем).

Последние 4 функции API предназначены для управления отображаемыми на память файлами. Чтобы отобразить файл, сначала нужно создать объект отображения файла с помощью функции `CreateFileMapping`. Эта функция возвращает описатель объекта отображения файла и иногда еще и вводит в файловую систему имя для этого объекта, чтобы другой процесс мог его использовать. Следующие две функции соответственно создают и удаляют отображение файлов. Последняя функция нужна для того, чтобы отобразить файл, который в данный момент отображен другим процессом. Таким образом, два и более процессов могут совместно использовать части своих адресных пространств.

Эти функции API являются основными. На них строится вся остальная система управления памятью. Например, существуют функции API для размещения и освобождения структур данных в одной или нескольких кучах. Кучи используются для хранения структур данных, которые динамически создаются и разрушаются. Кучи не освобождаются в процессе уборки мусора, поэтому пользовательское программное обеспечение само должно освобождать блоки виртуальной памяти, которые уже не нужны (уборкой мусора называют автоматическое удаление неиспользуемых структур данных). Куча в Windows 7 напоминает результат вызова функции `malloc` в UNIX, но в Windows XP, в отличие от UNIX, может быть несколько независимых куч.

Примеры виртуального ввода-вывода

Любая операционная система в первую очередь предназначена для обслуживания пользовательских программ, а основными услугами является ввод-вывод файлов. И UNIX, и Windows XP предлагают широкий спектр услуг ввода-вывода. Для большинства системных вызовов UNIX в Windows 7 имеется эквивалентный вызов, но обратное неверно, поскольку Windows 7 поддерживает гораздо больше вызовов и каждый из них гораздо сложнее соответствующего вызова UNIX.

Виртуальный ввод-вывод в UNIX

Система UNIX весьма популярна во многом благодаря своей простоте, которая, в свою очередь, является прямым результатом организации файловой системы.

Обычный файл представляет собой линейную последовательность 8-разрядных байтов[1] от 0 до $2^{64}-1$ максимум. Сама операционная система не подразумевает никакой конкретной структуры записей в файлах, хотя многие пользовательские программы рассматривают текстовые файлы в коде ASCII как последовательности строк, каждая из которых завершается символом перевода строки.

С каждым открытым файлом связан указатель на следующий байт, который нужно считать или записать. Системные вызовы `read` и `write` считывают и записывают данные, начиная с позиции, которую определяет указатель. После операции оба вызова перемещают указатель в другую позицию, передвигая его ровно на то количество байтов, которое было считано или записано. Возможен и произвольный доступ к файлам, когда файловый указатель устанавливается на заданное значение.

Помимо обычных файлов, система поддерживает специальные файлы, которые используются для доступа к устройствам ввода-вывода. С каждым устройством ввода-вывода обычно связан один или несколько специальных файлов. Считывая информацию из этих файлов и записывая информацию в эти файлы, программа может получать информацию от устройства ввода-вывода и выводить информацию на устройство ввода-вывода. Так происходит работа с дисками, принтерами, терминалами и многими другими устройствами.

Основные системные вызовы для файлов в UNIX приведены в табл. 6.5. Вызов `creat` (без привычной буквы *e* в конце) используется для создания нового файла. В настоящее время он не обязателен, поскольку вызов `open` тоже ведет к созданию нового файла. Вызов `unlink` удаляет файл (предполагается, что файл существует только в одном каталоге).

Таблица 6.5. Основные системные вызовы UNIX

Системный вызов	Описание
creat(name, mode)	Создание файла; параметр mode определяет режим защиты
unlink(name)	Удаление файла (предполагается, что связь у него единственная)
open(name, mode)	Открытие или создание файла и возвращение дескриптора файла
close(fd)	Закрытие файла
read(fd, buffer, count)	Считывание из файла байтов в buffer в количестве count
write(fd, buffer, count)	Запись в файл байтов из buffer в количестве count
lseek(fd, offset, w)	Перемещения файлового указателя на значения, заданные параметрами offset и w
stat(name, buffer)	Возвращение информации о файле
chmod(name, mode)	Изменение режима защиты файла
fcntl(fd, cmd, ...)	Выполнение различных управляющих операций (например, блокирование файла или его части)

Вызов `open` используется для открытия существующих файлов, а также для создания новых. Флаг `mode` сообщает, как его открывать (для чтения, для записи

[1] Для многих слова о 8-разрядных байтах могут показаться странными, однако раньше байт действительно мог быть и 5-, и 7-, и 8-разрядным. Теперь же мы по привычке считаем, что в байте ровно 8 бит. — *Примеч. науч. ред.*

и т. д.). Вызов возвращает небольшое целое число, которое называется **дескриптором файла**. Дескриптор файла идентифицирует файл при последующих вызовах.

Сам процесс ввода-вывода осуществляется вызовами `read` и `write`, каждый из которых в качестве параметров получает дескриптор файла (он указывает, какой файл использовать), буфер для данных и число передаваемых байтов. Вызов `lseek` используется для перемещения файлового указателя, что делает возможным доступ к произвольному месту в файле.

Вызов `stat` возвращает информацию о файле (размер, время последнего доступа, имя владельца и т. п.), вызов `chmod` изменяет режим защиты файла (например, разрешает или, наоборот, запрещает каким-нибудь пользователям читать его), наконец, вызов `fcntl` позволяет выполнять различные действия с файлом, например блокировать или разблокировать.

В листинге 6.3 показано, как происходит процесс ввода-вывода. Эта минимальная по объему программа, которая не включает в себя код проверки ошибок. Перед тем как войти в цикл, программа открывает существующий файл `data` и создает новый файл `newf`. Каждый вызов возвращает дескриптор файла `infd` или `outfd` соответственно. Следующий параметр в обоих вызовах — биты защиты, которые определяют, что файлы нужно считать и записать соответственно. Оба вызова возвращают дескриптор файла. Если вызов `open` или `creat` оказывается неудачным, возвращается отрицательный дескриптор файла.

Листинг 6.3. Фрагмент программы для копирования файла с использованием системных вызовов UNIX. Этот фрагмент написан на языке C, поскольку в языке Java нельзя показать необходимые нам низкоуровневые системные вызовы

```
/* Получение дескриптора файла. */
infd = open("data", 0);
outfd = creat("newf", ProtectionBits);
/* Цикл копирования. */
do {
        count = read(infd, buffer, bytes);
        if (count > 0) write(outfd, buffer, count);
} while (count > 0);

/* Закрытие файлов.*/
close(infd);
close(outfd);
```

Вызов `read` имеет три параметра: дескриптор файла, буфер и число байтов. Данный вызов должен считать нужное число байтов из указанного файла в буфер. Число считанных байтов помещается в переменную `count`. Значение `count` может быть меньше, чем `bytes`, если файл окажется слишком коротким. Вызов `write` копирует считанные байты в выходной файл. Цикл продолжается до тех пор, пока входной файл не будет прочитан полностью. Тогда цикл завершается, и оба файла закрываются.

Дескрипторы файлов в UNIX представляют собой небольшие целые числа (обычно до 20). Дескрипторы файлов 0, 1 и 2 соответствуют **стандартному вводу**, **стандартному выводу** и **стандартной ошибке** соответственно. Обычно первый из них относится к клавиатуре, а второй и третий — к дисплею, хотя пользователь может перенаправить любой стандартный поток ввода-вывода

в файл. Многие программы UNIX получают входные данные из стандартного ввода и записывают выходные данные в стандартный вывод. Такие программы называются **фильтрами**.

С файловой системой тесно связана система каталогов. Каждый пользователь может иметь несколько каталогов, а каждый каталог может содержать файлы и вложенные каталоги. Система UNIX обычно конфигурируется с главным каталогом, так называемым **корневым каталогом**, который содержит вложенные каталоги `bin` (для часто используемых программ), `dev` (для специальных файлов устройств ввода-вывода), `lib` (для библиотек) и `usr` (для пользовательских каталогов, как показано на рис. 6.29). В нашем примере каталог `usr` содержит вложенные каталоги `ast` и `jim`. Каталог `ast` включает в себя два файла (`data` и `foo.c`) и вложенный каталог `bin`, в который входят 4 файла (`game1`, `game2`, ...).

Для обращения к файлу нужно указать его **путь** из корневого каталога. Путь содержит список всех каталогов от корневого каталога к файлу, для разделения каталогов используется косая черта (`/`). Например, путь к файлу `game2` выглядит следующим образом: `/usr/ast/bin/game2`. Путь, который начинается с корневого каталога, называется **абсолютным**.

В каждый момент времени каждая работающая программа имеет **текущий каталог**. Путь может быть связан с текущим каталогом. В этом случае в начале пути косая черта не ставится (чтобы отличить от абсолютного пути). Такой путь называется **относительным путем**. Если `/usr/ast` — текущий каталог, то можно получить доступ к файлу `game3`, используя путь `bin/game3`. Пользователь может создать **связь** с чужим файлом, используя для этого системный вызов `link`. В нашем примере пути `/usr/ast/bin/game3` и `/usr/jim/jotto` приводят к одному и тому же файлу. Не разрешается применять связи к каталогам, чтобы предотвратить циклы в системе каталогов. В вызовах `open` и `creat` могут использоваться как абсолютные, так и относительные пути.

Основные вызовы для манипулирования каталогами UNIX приведены в табл. 6.6. Вызов `mkdir` создает новый каталог, а `rmdir` удаляет существующий пустой каталог. Следующие три вызова применяются для чтения элементов каталогов. Первый открывает каталог, второй считывает элементы из него, третий закрывает каталог. Вызов `chdir` изменяет текущий каталог.

Таблица 6.6. Основные вызовы для работы с каталогами в системе UNIX

Системный вызов	Описание
mkdir(name, mode)	Создание нового каталога
rmdir(name)	Удаление пустого каталога
Opendir(name)	Открытие каталога для чтения
readdir(dirpointer)	Считывание следующего элемента каталога
Closedir(dirpointer)	Закрытие каталога
chdir(dirname)	Смена текущего каталога на каталог с именем dirname
link(name1, name2)	Создание связи (элемента каталога name2, указывающего на каталог name1)
unlink(name)	Удаление связи (элемента name) из каталога

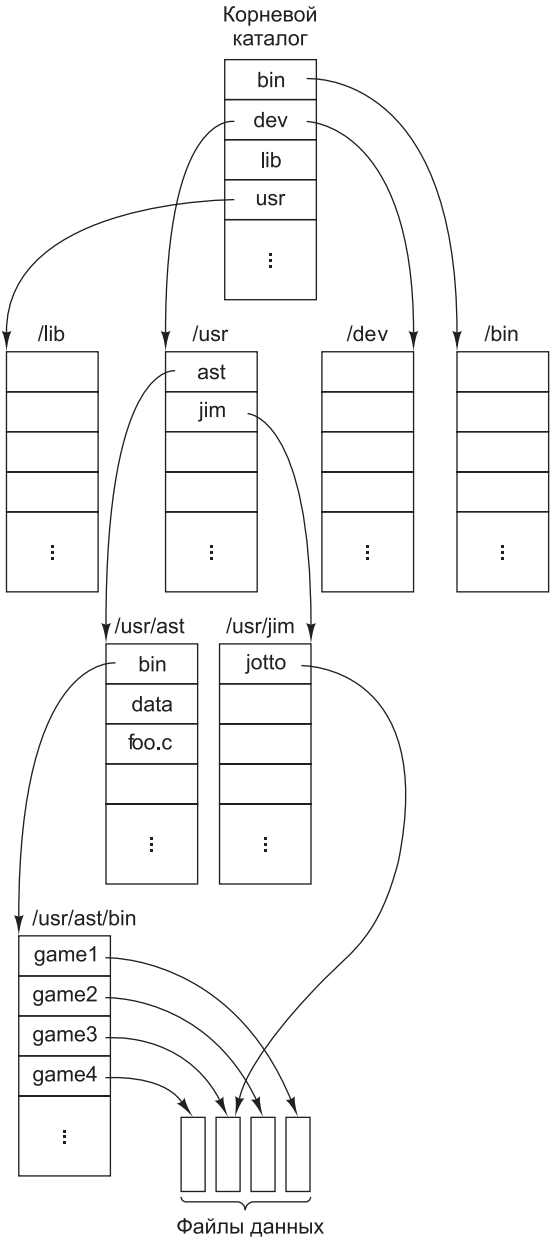


Рис. 6.29. Фрагмент системы каталогов операционной системы UNIX

Вызов `link` создает элемент каталога, который указывает на уже существующий файл. Например, элемент `/usr/jim/jotto` можно создать вызовом:

```
link("/usr/ast/bin/game3", "/usr/jim/jotto")
```

Того же результата можно добиться эквивалентным вызовом с относительным путем, который зависит от текущего каталога. Вызов `unlink` удаляет элемент

каталога. Если файл имеет только одну связь, он удаляется. Если файл имеет две и более связей, то он не удаляется. Не имеет никакого значения, была ли удаленная связь создана изначально или это копия. Следующий вызов делает файл `game3` доступным только через путь `/usr/jim/jotto`:

```
unlink("/usr/ast/bin/game3")
```

Вызовы `link` и `unlink` могут использоваться для «перемещения» файлов из одного каталога в другой.

С каждым файлом (а также с каждым каталогом, поскольку каталог — это тоже файл) связана битовая карта, которая сообщает, кому разрешен доступ к файлу. Карта содержит три поля RWX (Read, Write, eXecute — чтение, запись, выполнение). Первое из них контролирует разрешение на чтение, запись и выполнение файлов для их владельца, второе — для других пользователей из группы владельца, третье — для всех остальных пользователей. Например, биты RWX R-X --X означают, что владелец файла может читать этот файл, записывать что-либо в него и выполнять его (очевидно, файл является исполняемой программой, иначе не было бы разрешения на его выполнение), другие члены группы могут читать и выполнять его, а все остальные — только выполнять. Таким образом, посторонние пользователи смогут выполнить эту программу, но не смогут ее украсть (скопировать), поскольку им запрещено чтение. Включение пользователей в те или иные группы осуществляется системным администратором, которого обычно называют **привилегированным пользователем**. Привилегированный пользователь имеет право действовать вопреки механизму защиты и считывать, записывать и выполнять любой файл.

Теперь посмотрим, как файлы и каталоги реализованы в системе UNIX. За более подробным описанием обращайтесь к [Vahalia, 1996]. С каждым файлом (и с каждым каталогом, поскольку каталог — это тоже файл) связан информационный блок размером 64 байта, который называется **индексным дескриптором** (i-node). Индексный дескриптор содержит информацию о том, кто владеет файлом, что разрешено делать с файлом, где найти данные и. т. п. Индексные дескрипторы для файлов расположены либо последовательно в начале диска, либо, если диск разделен на группы цилиндров, — в начале группы. Индексные дескрипторы снабжены последовательными номерами. Таким образом, система UNIX может обнаружить индексный дескриптор просто путем вычисления его адреса на диске.

Элемент каталога состоит из двух частей: имени файла и номера индексного дескриптора. Когда программа выполняет следующую команду, система ищет текущий каталог файла `foo.c`, чтобы найти номер индексного дескриптора этого файла:

```
open("foo.c", 0)
```

Обнаружив номер индексного дескриптора, программа может считать его и узнать всю информацию о файле.

При большей длине пути файла процедура повторяется несколько раз, пока не будет пройден весь путь. Например, чтобы найти номер индексного дескриптора для пути `/usr/ast/data`, система сначала отыщет корневой каталог для элемента `usr`. Обнаружив индексный дескриптор `usr`, она может прочитать файл (каталог в системе UNIX — это тоже файл). В этом файле она отыщет элемент `ast` и най-

дет номер индексного дескриптора для файла `/usr/ast`. Считав информацию о местонахождении каталога `/usr/ast`, система сможет обнаружить элемент для `data` и, следовательно, — номер i-node для `/usr/ast/data`. Найдя номер индексного дескриптора для этого файла, система может получить любую информацию об этом файле.

Формат, содержание и размещение индексных дескрипторов несколько различаются в разных системах (особенно когда идет речь о сетевых системах), но следующие элементы присутствуют практически в каждом индексном дескрипторе:

- тип файла, три поля RWX (всего 9 бит) и некоторые другие биты;
- число связей с файлом (число элементов каталогов);
- идентификатор владельца;
- группа владельца;
- длина файла в байтах;
- тринадцать дисковых адресов;
- время, когда файл читали в последний раз;
- время, когда последний раз производилась запись в файл;
- время, когда в последний раз менялся индексный дескриптор.

Возможные типы файлов — обычные файлы, каталоги, два вида особых файлов для блочных и неструктурированных устройств ввода-вывода. Число связей и идентификатор владельца мы уже обсуждали. Длина файла выражается 32-разрядным целым числом, обозначающим старший байт файла. Вполне возможно создать файл, перенести указатель в позицию 1 000 000 и записать 1 байт. В результате получится файл длиной 1 000 001. При этом файл *не* потребует сохранения всех «несуществующих» байтов.

Первые 10 адресов на диске указывают на блоки данных. Если размер блока — 1024 байт, то можно работать с файлами размером до 10 240 байт. Адрес 11 указывает на **блок косвенной адресации**, который содержит дополнительные адреса. С 1024-байтовым блоком и 32-разрядными адресами он содержит 256 адресов. Это позволяет работать с файлами размером до 10 240 + 256 × 1024 = = 272 384 байта. Для файлов еще большего размера существует адрес 12, который указывает на 256 блоков косвенной адресации. Здесь допустимый размер файлов составляет 272384 + 256 × 256 × 1024 = 67 381 248 байт. Если и эта схема **блока двойной косвенной адресации** слишком мала, используется адрес 13. Он указывает на **блок тройной косвенной адресации**, который содержит адреса 256 блоков двойной косвенной адресации. Используя прямую, косвенную, двойную косвенную и тройную косвенную адресацию, можно обращаться к 16 843 018 блокам. Это значит, что максимально возможный размер файла составляет 17 247 250 432 байта. Если вместо 32-разрядных адресов используются 64-разрядные, а размер блока составляет 4 Кбайт, файлы могут быть очень, *очень* большими. Свободные дисковые блоки хранятся в виде связного списка. Если нужен новый блок, он берется из списка. В результате получается, что блоки каждого файла беспорядочно раскиданы по всему диску.

Для повышения скорости дискового ввода-вывода при открытии файла его индексный дескриптор копируется в таблицу основной памяти и хранится там, пока файл остается открытым. Кроме того, в памяти хранится набор блоков,

к которым недавно производилось обращение. Так как большинство файлов считывается последовательно, часто при обращении к файлу требуется тот же блок, что и при предыдущем обращении. Чтобы увеличить скорость, система считывает *следующий* блок в файл еще до того, как к нему произведено обращение. Все эти моменты скрыты от пользователя. Когда пользователь выдает вызов `read`, программа приостанавливается, пока требуемые данные не появятся в буфере.

Зная все это, можно понять, как происходит процесс ввода-вывода. Вызов `open` заставляет систему искать каталоги по определенному пути. Если поиск оказывается успешным, индексный дескриптор считывается во внутреннюю таблицу. Вызовы `read` и `write` требуют, чтобы система вычислила номер блока из текущей позиции файла. Адреса первых 10 дисковых блоков всегда находятся в основной памяти (в индексном дескрипторе); для остальных блоков сначала требуется считать один или несколько блоков косвенной адресации. Вызов `lseek` просто меняет текущую позицию указателя и не производит никакого ввода-вывода.

Также несложно понять и реализацию вызовов `link` и `unlink`. В первом аргументе вызова `link` находится номер индексного дескриптора. По этому номеру он создает элемент каталога для второго аргумента и помещает номер i-node первого файла в этот элемент каталога. Наконец, он увеличивает число связей в индексном дескрипторе на единицу. Вызов `unlink` удаляет элемент каталога и уменьшает число связей в индексном дескрипторе. Если это число становится равным 0, файл удаляется, и все его блоки помещаются в список свободных блоков.

Виртуальный ввод-вывод в Windows 7

Windows XP поддерживает несколько файловых систем, самые важные из которых — **NTFS** (NT File System — файловая система NT) и **FAT** (File Allocation Table — таблица размещения файлов). Первая была разработана специально для Windows XP. Вторая является устаревшей файловой системой для MS-DOS, которая также используется в Windows 95/98 (хотя и с длинными именами файлов). Поскольку система FAT устарела и сейчас встречается только в USB-дисках и картах памяти, мы рассмотрим только файловую систему NTFS.

В NTFS имя файла может быть длиной до 255 символов. Имена файлов написаны в кодировке Unicode, благодаря чему люди в разных странах, где не используется латинский алфавит, могут писать имена файлов на их родном языке. Более того, Unicode используется во внутренней реализации Windows 7; во всех версиях операционной системы, начиная с Windows 2000, тексты меню, сообщений об ошибках и прочих элементов интерфейса хранятся в выделенных для каждого языка конфигурационных файлах, в то время как двоичные файлы едины для всех вариантов языкового окружения. В NTFS прописные и строчные буквы в именах файлов считаются разными (то есть «foo» отличается от «FOO»). К сожалению, в Win32 API прописные и строчные буквы в именах файлов и каталогов не различаются, поэтому это преимущество теряется для программ, использующих подсистему Win32.

Как и в UNIX, файл представляет собой линейную последовательность байтов, максимальная длина которой составляет $2^{64} - 1$. Указатели тоже существуют, но их длина не 32, а 64 бита, чтобы можно было поддерживать максимальную длину файла. Вызовы функций в Win32 API для манипуляций с каталогами и файлами в целом напоминают вызовы функций в UNIX, но большинство из

них имеют больше параметров и другую модель защиты. При открытии файла возвращается описатель, который затем используется для чтения и записи файла. В отличие от UNIX описатели не являются маленькими целыми числами, потому что они используются для идентификации объектов ядра, которые могут исчисляться миллионами. Основные функции Win32 API для управления файлами перечислены в табл. 6.7.

Таблица 6.7. Основные функции Win32 API для ввода-вывода файлов

Функция API	Аналог в UNIX	Описание
CreateFile	open	Создание файла или открытие существующего файла; функция возвращает описатель
DeleteFile	unlink	Удаление существующего файла
CloseHandle	close	Закрытие файла
ReadFile	read	Считывание данных из файла
WriteFile	write	Запись данных в файл
SetFilePointer	lseek	Установка файлового указателя на заданное место в файле
SetFileAttributes	stat	Возвращение свойств файла
LockFile	Fcntl	Блокирование области файла, чтобы обеспечить взаимное исключение доступа
UnlockFile	Fcntl	Разблокирование ранее заблокированной области файла

Рассмотрим эти вызовы. Вызов `CreateFile` используется для создания нового файла и возвращает описатель для них. Эта функция применяется также для открытия уже существующего файла, поскольку в API нет функции `open`. Мы не будем приводить параметры функций API, поскольку их очень много. Например, функция `CreateFile` имеет семь параметров:

- указатель на имя создаваемого или открываемого файла;
- флаги, которые сообщают, какие действия разрешено производить с файлом (читать, записывать или то и другое);
- флаги, которые показывает, могут ли несколько процессов открывать файл одновременно;
- указатель на дескриптор безопасности, который показывает, кто имеет доступ к файлу;
- флаги, которые показывает, что делать, когда файл существует или не существует;
- флаги, связанные с атрибутами архивации, компрессии и т. д.;
- описатель файла, атрибуты которого нужно клонировать для нового файла.

Следующие шесть функций API похожи на соответствующие функции UNIX. Однако следует учитывать, что ввод-вывод в Windows 7 является асинхронным, хотя процесс может дожидаться завершения операции. Последние две функции позволяют блокировать или разблокировать область файла, чтобы гарантировать процессам исключение доступа к одной и той же области.

Используя эти функции API, можно написать процедуру копирования файла, аналогичную процедуре из листинга 6.3. Такая процедура (без кода проверки ошибок) представлена в листинге 6.4. На практике программу для копирования файла писать не нужно, поскольку существует функция `CopyFile`, которая делает примерно то же самое и реализована в виде библиотечной процедуры.

Листинг 6.4. Фрагмент программы для копирования файла с помощью функции API системы Windows 7. Этот фрагмент написан на языке C, поскольку в языке Java нельзя показать необходимые нам низкоуровневые системные вызовы

```
/* Открытие файлов для ввода и вывода. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL,
  OPEN_EXISTING, 0, NULL);
outhandle = CreateFileC'newF, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
  FILE_ATTRIBUTE_NORMAL, NULL);

/* Копирование файла. */
do {
  s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
  if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Закрытие файлов. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Windows 7 поддерживает иерархическую файловую систему, напоминающую файловую систему UNIX. Однако в качестве разделителя здесь используется не прямая, а обратная косая черта (заимствовано из MS-DOS). Здесь тоже существует понятие текущего каталога, а пути могут быть абсолютными и относительными. Однако между Windows XP и UNIX есть одно существенное различие. UNIX позволяет монтировать в одно дерево файловые системы с разных дисков и машин, скрывая таким образом от программ структуру диска. Windows XP такой возможности не имеет, поэтому абсолютные имена файлов должны начинаться с буквы диска (например, C:\windows\system\foo.dll). В то же время, начиная с Windows 2000, реализована возможность монтирования файловых систем в стиле UNIX.

Основные функции для работы с каталогами перечислены в табл. 6.8 (также вместе с эквивалентами из UNIX). Вероятно, смысл этих функций не требует пояснений.

Таблица 6.8. Основные функции Win32 API для работы с каталогами

Функция API	Функция UNIX	Значение
CreateDirectiry	mkdir	Создание нового каталога
RemoveDirectory	rmdir	Удаление пустого каталога
FindFirstFile	opendir	Начало чтения элементов каталога
FindNextFile	readdir	Чтение следующего элемента каталога
MoveFile		Перемещение файла из одного каталога в другой
SetCurrentDirectory	chdir	Смена текущего каталога

Windows XP имеет более сложный механизм защиты, чем UNIX. Хотя API содержит сотни функций, связанных с безопасностью, следующее описание дает хотя бы приблизительное представление. Когда пользователь входит в систему, его процесс получает **маркер доступа** от операционной системы. Маркер доступа содержит **идентификатор безопасности** (Security ID, SID), список групп, к которым принадлежит пользователь, имеющиеся привилегии и некоторую другую информацию. Маркер доступа концентрирует всю информацию о защите в одном легко доступном месте. Все процессы, созданные этим процессом, наследуют этот же маркер доступа.

Дескриптор безопасности (security descriptor) — это один из параметров, который дается любому объекту при его создании. Дескриптор безопасности содержит список элементов, который называется **списком контроля доступа** (Access Control List, **ACL**). Каждый элемент разрешает или запрещает совершать определенный набор операций с объектом какому-либо пользователю или группе. Например, файл может содержать дескриптор безопасности, который определяет, что Иванов не имеет доступа к файлу вообще, Петров может читать файл, Сидоров может читать и записывать файл, а все члены группы XYZ могут считать только размер файла.

Если процесс пытается выполнить какую-либо операцию с объектом, используя описатель, который он получил при открытии объекта, диспетчер безопасности получает маркер доступа данного процесса и сравнивает уровень целостности дескриптора безопасности объекта с уровнем целостности маркера. Процесс не может получить маркер с разрешениями записи для объекта с более высоким уровнем целостности. Уровни целостности в основном используются для ограничения возможностей модификации системы кодом, загруженным в веб-браузере. После проверки уровня целостности диспетчер безопасности начинает перебирать элементы списка контроля доступа по порядку. Как только он находит элемент, который соответствует нужному пользователю или одной из групп, найденная информация о разрешении или запрещении доступа принимается в качестве заданной. По этой причине элементы, запрещающие доступ, обычно помещаются в список контроля доступа перед элементами, разрешающими доступ (чтобы пользователь, у которого нет доступа, не смог получить его незаконно, будучи членом одной из групп, которой доступ разрешен). Дескриптор безопасности также содержит информацию, используемую для аудита доступов к объекту.

А теперь давайте в общих чертах рассмотрим реализацию файлов и каталогов в Windows 7. Каждый диск разделен на тома, такие же, как дисковые разделы UNIX. Каждый том содержит файлы, битовые карты каталогов и другие структуры данных. Каждый том организован в виде линейной последовательности **кластеров**. Размер кластера фиксирован для каждого тома. Он может составлять от 512 байт до 64 Кбайт, в зависимости от размера тома. Обращение к кластеру осуществляется по смещению от начала тома. При этом используются 64-разрядные числа.

Основной структурой данных в каждом томе является **главная файловая таблица** (Master File Table, **MFT**), в которой содержатся записи для каждого файла и каталога в томе. Эти записи аналогичны элементам индексного дескриптора (i-node) в UNIX. Главная файловая таблица является файлом и может быть помещена в любое место в пределах тома. Это устраняет проблему, возникающую при обнаружении испорченных дисковых блоков среди индексных дескрипторов.

Главная файловая таблица показана на рис. 6.30. Она начинается с заголовка, в котором дается информация о томе (указатели на корневой каталог, файл

загрузки, список лиц, пользующихся свободным доступом и т. д.). Затем идет по одному элементу на каждый файл или каталог (1 Кбайт за исключением тех случаев, когда размер кластера составляет 2 Кбайт и более). Каждый элемент содержит все метаданные (административную информацию) о файле или каталоге. Допускается несколько форматов, один из которых показан на рис. 6.30.

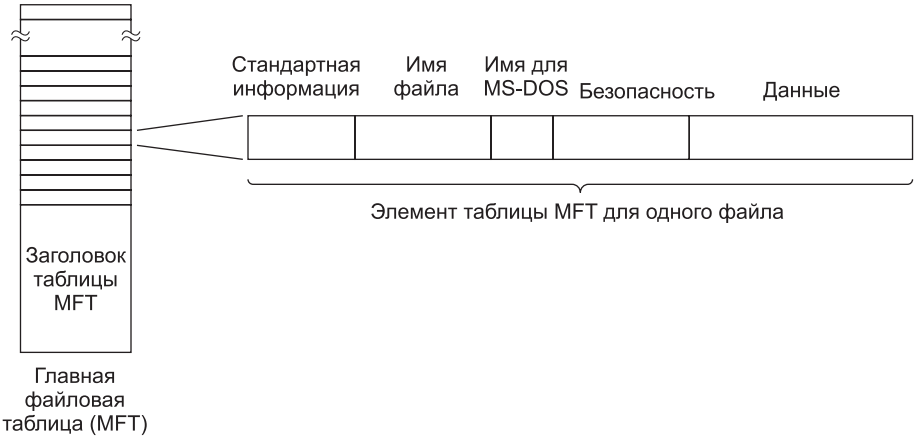


Рис. 6.30. Главная файловая таблица в системе Windows 7

Поле стандартной информации содержит информацию меток времени, необходимых стандартах POSIX, числе связей, битах «только чтение», битах архивирования и т. д. Это поле имеет фиксированную длину и является обязательным. Имя файла может иметь любую длину в пределах 255 Unicode-символов. Чтобы такие файлы стали доступными для устаревших 16-разрядных программ, они могут снабжаться дополнительным именем MS-DOS, состоящим максимум из 8 символов, за которыми следует точка и расширение из трех символов. Если реальное имя файла соответствует правилу именования MS-DOS (8 + 3), второе имя в стиле MS-DOS не используется.

Далее следует поле безопасности. Во всех версиях, вплоть до Windows NT 4.0, в поле безопасности содержался дескриптор безопасности. Начиная с Windows 2000, вся информация о безопасности помещается в один файл, а поле безопасности просто указывает на соответствующую часть этого файла.

Для файлов небольшого размера сами данные этих файлов могут содержаться в записи главной файловой таблицы, что упрощает их вызов — для этого не требуется обращаться к диску. Данная концепция получила название **непосредственный файл** [Mullender and Tanenbaum, 1984]. Для файлов большого размера это поле содержит указатели на кластеры с данными или (что более распространено) на блоки последовательных кластеров, так что номер кластера и его длина могут представлять произвольное количество данных. Если элемент главной файловой таблицы недостаточно велик для хранения нужной информации, с ним можно связать один или несколько дополнительных элементов (записей).

Максимальный размер файла составляет 2^{64} байт. Поясним, что собой представляет файл такого размера. Представим, что файл записан в двоичной системе, а каждый 0 или 1 занимает 1 мм пространства. Величина 2^{67} мм соответствует значению 15 световых лет. Этого хватило бы для того, чтобы выйти за пределы Солнечной системы, достичь альфы Центавра и вернуться обратно.

Файловая система NTFS имеет много других интересных особенностей, в частности она поддерживает компрессию данных и механизм отказоустойчивости на основе атомарных транзакций. Дополнительную информацию можно найти в [Russinovich and Solomon, 2005].

Примеры управления процессами

Системы Windows 7 и UNIX позволяют разделить задание на несколько процессов, выполняющихся (псевдо-)параллельно и взаимодействующих друг с другом, как в примере с производителем и потребителем, который мы обсуждали ранее. В этом подразделе мы поговорим о том, как происходит управление процессами в обеих системах. Обе системы поддерживают параллелизм в пределах одного процесса с использованием программных потоков, и об этом мы тоже расскажем.

Управление процессами в UNIX

В любой момент процесс в UNIX может создать субпроцесс, являющийся его точной копией. Для этого выполняется системный вызов `fork`. Исходный процесс называется **родительским**, а новый — **дочерним**. Два процесса, полученные в результате вызова `fork`, абсолютно идентичны и даже разделяют одни и те же файловые дескрипторы. Однако каждый из этих двух процессов выполняет свою работу независимо от другого.

Часто дочерний процесс определенным образом манипулирует с дескрипторами файлов, а затем выполняет системный вызов `exec`, который заменяет программу и данные программой и данными из выполняемого файла, определенного в качестве параметра вызова `exec`. Например, если пользователь вводит команду `xyz`, то интерпретатор команд (оболочка) выполняет операцию `fork`, порождая таким образом дочерний процесс. А этот процесс выполняет вызов `exec`, чтобы запустить программу `xyz`.

Эти два процесса работают параллельно (с системным вызовом `exec` или без него), но иногда родительский процесс должен по каким-либо причинам ждать, чтобы дочерний процесс завершил свою работу, и только после этого продолжает выполнение тех или иных действий. В этом случае родительский процесс выполняет системный вызов `wait` или `waitpid`, в результате чего он временно приостанавливается и ждет, пока дочерний процесс не выполнит системный вызов `exit`.

Процессы могут выполнять вызов `fork` сколь угодно часто, в результате чего получается целое дерево процессов. Посмотрите на рис. 6.31. Здесь процесс *A* выполнил вызов `fork` дважды и породил два новых процесса, *B* и *C*. Затем процесс *B* тоже выполнил вызов `fork` дважды, а процесс *C* — один раз. Таким образом, получилось дерево из шести процессов.

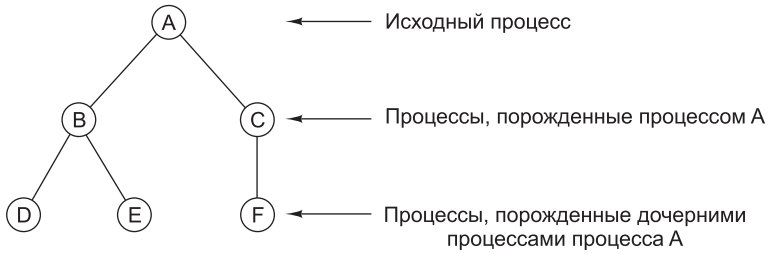


Рис. 6.31. Дерево процессов в системе UNIX

Процессы в UNIX могут взаимодействовать друг с другом через специальную информационную структуру, которая называется **каналом**. Канал представляет собой вид буфера, в который один процесс записывает поток данных, а другой процесс извлекает оттуда эти данные. Байты всегда возвращаются из канала в том порядке, в котором они были записаны. Случайный доступ невозможен. Каналы не сохраняют границ между фрагментами данных, поэтому если один процесс записал в канал 4 фрагмента по 128 байт, а другой процесс считывает данные по 512 байт, то второй процесс получит все данные сразу без указания на то, что они были записаны за несколько приемов.

В System V и Linux применяется другой метод взаимодействия процессов. Здесь используются так называемые **очереди сообщений**. Процесс может создать новую очередь сообщений или открыть уже существующую с помощью вызова `msgget`. Для отправки сообщений используется вызов `msgsnd`, а для получения — `msgrecv`. Сообщения, отправленные таким способом, отличаются от данных, помещаемых в канал. Во-первых, границы сообщений сохраняются, в то же время канал просто передает поток байтов. Во-вторых, сообщения имеют приоритеты, поэтому срочные сообщения идут перед всеми остальными. В-третьих, сообщения типизированы, и вызов `msgrecv` позволяет определять их тип, если это необходимо.

Два и более процессов могут разделять общую область своих адресных пространств. UNIX управляет этой разделенной памятью путем отображения одних и тех же страниц на виртуальное адресное пространство всех разделенных процессов. В результате запись в общую область, произведенная одним из процессов, будет видна всем остальным процессам. Этот механизм обеспечивает очень высокую пропускную способность при взаимодействии процессов. Системные вызовы, задействованные в работе с общей памятью, называются `shmat` и `shmop`.

Еще одна особенность System V и Linux — наличие семафоров. Принципы их работы мы уже описывали в примере с производителем и потребителем.

Еще одно свойство POSIX-совместимых систем UNIX — поддержка нескольких программных потоков управления в пределах одного процесса. Эти потоки управления обычно называют просто **программными потоками**. Они похожи на процессы, которые совместно используют общее адресное пространство и все объекты, связанные с этим адресным пространством (дескрипторы файлов, переменные окружения и т. д.). Однако каждый поток имеет собственный счетчик команд, собственные регистры и собственный стек. Если один из программных потоков приостанавливается (например, пока не завершится процесс ввода-вывода), другие программные потоки в том же процессе могут продолжать работу. Два программных потока в одном процессе, которые действуют как процесс-производитель и процесс-потребитель, напоминают два однопоточных процесса, которые разделяют сегмент памяти, содержащий буфер, хотя не идентичны им. Во втором случае каждый процесс имеет собственные дескрипторы файлов и т. д., тогда как в первом случае все эти элементы общие. В примере взаимодействия производителя и потребителя мы видели, как работают потоки в Java. Иногда для поддержки каждого из программных потоков исполняющая система Java использует программный поток операционной системы, однако это не обязательно.

В каких случаях могут понадобиться программные потоки? Например, веб-сервер может хранить в основной памяти кэш часто используемых веб-страниц. Если нужная страница находится в кэш-памяти, то она выдается немедленно.

Если нет, то она вызывается с диска. К сожалению, на это требуется довольно длительное время (обычно 20 миллисекунд), на это время процесс блокируется и не может обслуживать вновь поступающие запросы, даже если запрашиваемые веб-страницы находятся в кэш-памяти.

Для решения проблемы можно создать несколько программных потоков в одном процессе, которые совместно используют общий кэш веб-страниц. Если один из программных потоков блокируется, новые запросы могут обрабатываться другими программными потоками. Конечно, предотвратить блокировку процессов можно и без использования программных потоков. Для этого потребуется несколько процессов, но тогда придется продублировать кэш, а это является неэффективным использованием памяти.

Стандарт системы UNIX на программные потоки называется **pthreads** и определяется в POSIX (P1003.1C). Он описывает вызовы для управления программными потоками и их синхронизации. В стандарте ничего не сказано о том, должно ли управлять программными потоками ядро, или они должны функционировать только в пользовательском пространстве. Наиболее распространенные функции для работы с программными потоками приведены в табл. 6.9.

Таблица 6.9. Основные функции для работы с программными потоками, определенные в стандарте POSIX

Функция	Описание
pthread_create	Создание нового программного потока в адресном пространстве вызывающей процедуры
pthread_exit	Завершение программного потока
pthread_join	Ожидание завершения программного потока
pthread_mutex_init	Создание нового мьютекса
pthread_mutex_destroy	Удаление мьютекса
pthread_mutex_lock	Блокирование мьютекса
pthread_mutex_unlock	Разблокирование мьютекса
pthread_cond_init	Создание условной переменной
pthread_cond_destroy	Удаление условной переменной
pthread_cond_wait	Ждет условную переменную
pthread_cond_signal	Разблокирование одного из программных потоков, ожидающего условной переменной

Давайте рассмотрим эти вызовы. Первый вызов, `pthread_create`, создает новый программный поток. После выполнения этой процедуры в адресном пространстве появляется на один программный поток больше. Программный поток, который выполнил свою работу, вызывает функцию `pthread_exit`. Если потоку нужно подождать, пока другой поток окончит работу, он вызывает функцию `pthread_join`. Если этот другой программный поток уже закончил свою работу, вызов `pthread_join` немедленно завершается. В противном случае он блокируется.

Программные потоки можно синхронизировать с помощью специальных объектов, которые называются **мьютексами** (mutexes). Обычно мьютекс управляет

каким-либо ресурсом (например, буфером, разделенным между двумя программными потоками). Для того чтобы в конкретный момент времени только один поток мог получать доступ к общему ресурсу, потоки должны блокировать мьютекс перед использованием ресурса и разблокировать его после завершения работы с ним. Так можно избежать состояния гонок, поскольку этому протоколу подчиняются все программные потоки. Мьютексы похожи на бинарные семафоры (то есть семафоры, которые могут принимать только два значения: 0 или 1).

Мьютексы создаются и уничтожаются вызовами `pthread_mutex_init` и `pthread_mutex_destroy` соответственно. Мьютекс может находиться в одном из двух состояний: заблокированном и незаблокированном. Если программному потоку нужно заблокировать незаблокированный мьютекс, он выполняет вызов `pthread_mutex_lock`, а затем продолжает работу. Однако если программный поток попытается заблокировать уже заблокированный мьютекс, поток приостанавливается. Когда поток, который в данный момент использует общий ресурс, завершит работу с этим ресурсом, он должен разблокировать соответствующий мьютекс вызовом `pthread_mutex_unlock`.

Мьютексы предназначены для кратковременной блокировки (например, для защиты общей переменной), но не для долгосрочной синхронизации (например, для ожидания, пока освободится накопитель на магнитной ленте). Для длительной синхронизации существуют **условные переменные** (condition variables). Эти переменные создаются и удаляются вызовами `pthread_cond_init` и `pthread_cond_destroy` соответственно.

Условная переменная связана с двумя программными потоками: ожидающим и сигнальным. Если, например, программный поток обнаруживает, что накопитель на магнитной ленте, который ему нужен, в данный момент занят, этот поток выполняет для условной переменной вызов `pthread_cond_wait`. Когда поток, который использует накопитель на магнитной ленте, завершает свою работу с этим устройством (а это может произойти через несколько часов), он сигнализирует об этом вызовом `pthread_cond_signal`. Это позволяет разблокировать ровно один поток — тот, который ожидает эту условную переменную. При отсутствии программных потоков, ожидающих эту переменную, сигнал пропадает. У условных переменных нет счетчика, как у семафоров. Отметим, что с потоками, мьютексами и условными переменными можно выполнять и другие операции.

Управление процессами в Windows 7

Windows 7 поддерживает межпроцессные взаимодействия и синхронизации. Каждый процесс содержит, по крайней мере, один программный поток. Процессы и программные потоки в совокупности представляют собой инструменты управления параллелизмом в одно- и многопроцессорных системах.

Новые процессы создаются с помощью функции `CreateProcess` API. Эта функция имеет 10 параметров, у каждого из которых есть множество возможных значений. Очевидно, что такая система гораздо сложнее соответствующей схемы UNIX, где функция `fork` вообще не имеет аргументов, а у `exec` их всего три: указатели на имя выполняемого файла, на массив параметров командной строки и на строку описания конфигурации. Ниже перечислены 10 аргументов функции `CreateProcess`.

- указатель на имя исполняемого файла;
- сама командная строка (без синтаксического разбора);

- указатель на дескриптор безопасности данного процесса;
- указатель на дескриптор безопасности исходного программного потока;
- бит, который сообщает, наследует ли новый процесс описатели родительского процесса;
- различные флаги (например, ошибки, приоритета, отладки, консолей);
- указатель на строки описания конфигурации;
- указатель на имя рабочего каталога нового процесса;
- указатель на структуру, которая описывает исходное окно на экране;
- указатель на структуру, которая возвращает 18 значений вызывающей процедуре.

В Windows 7 не существует иерархии родительских дочерних процессов. Все процессы создаются равными. Однако поскольку одним из 18 параметров, возвращаемых исходному процессу, является описатель нового процесса (который дает возможность контролировать новый процесс), здесь существует внутренняя иерархия в том смысле, что определенным процессам доступны описатели других процессов. Эти описатели нельзя просто передать другому процессу, но процесс может сделать определенный описатель доступным для другого процесса, а затем передать ему этот описатель, так что внутренняя иерархия процессов не является постоянной.

Каждый процесс в Windows 7 создается с одним программным потоком, но позднее этот процесс может создать еще несколько таких потоков. Создать программный поток проще, чем процесс, поскольку вызов `CreateThread` имеет всего 6 параметров вместо 10: дескриптор безопасности, размер стека, начальный адрес, пользовательский параметр, начальное состояние потока (готов к работе или блокирован) и идентификатор потока. Поскольку созданием потоков занимается ядро, оно располагает информацией обо всех программных потоках (иными словами, их реализация не ограничивается пользовательским пространством, как в некоторых других системах).

В процессе планирования ядро вызывает не только процесс, который должен запускаться следующим, но и программный поток этого процесса. Это значит, что ядро всегда знает, какие программные потоки блокированы, а какие — нет. Так как программные потоки являются объектами ядра, они имеют дескрипторы безопасности и описатели. Поскольку описатели разрешается передавать другому процессу, можно сделать так, чтобы один процесс управлял программными потоками другого процесса. Эта особенность, например, пригодится для отладчиков.

Процессы могут взаимодействовать друг с другом разными способами: через каналы, именованные каналы, почтовые слоты, сокеты, удаленные вызовы процедур, общие файлы. Каналы делятся на два вида: байтовые каналы и каналы сообщений. Тип канала выбирается во время создания. Байтовые каналы работают так же, как в UNIX. Каналы сообщений сохраняют границы сообщений, поэтому четыре записи по 128 байт считываются из канала как четыре сообщения по 128 байт (а не как одно сообщение размером 512 байт, как в случае с каналами байтов). Кроме того, существуют именованные каналы, которые тоже бывают двух видов. Именованные каналы могут использоваться в сети, а обычные каналы — нет.

Сокеты похожи на каналы, но они обычно соединяют процессы на разных машинах, хотя их можно применять и для соединения процессов на одной машине.

Вообще говоря, соединение через сокет ненамного лучше связи через обычный или именованный канал.

Удаленные вызовы процедур позволяют процессу *A* дать процессу *B* команду совершить вызов процедуры в адресном пространстве *B* от имени *A* и возвратить результат процессу *A*. Здесь существуют различные ограничения на параметры вызова. Например, передача указателя другому процессу не имеет никакого смысла. Вместо этого приходится упаковывать и передавать другому процессу объект(ы), на который ссылается указатель.

Наконец, процессы могут совместно использовать общую память путем одновременного отображения на память одного и того же файла. Тогда все записи, порожденные одним процессом, появятся в адресном пространстве других процессов. Применяя такой механизм, можно легко реализовать общий буфер, который мы описывали в примере с процессом-производителем и процессом-потребителем.

Windows 7 предоставляет множество механизмов синхронизации (семафоры, мьютексы, критические секции, события). Все эти механизмы работают не с процессами, а с программными потоками, поэтому когда поток блокируется на семафоре, это никак не влияет на другие программные потоки этого процесса — они просто продолжают работать.

Семафор создается с помощью функции `CreateSemaphore` API, которая может установить его на определенное значение и определить его максимальное значение. Семафоры являются объектами ядра, поэтому они имеют дескрипторы безопасности и описатели. Описатель семафора можно продублировать с помощью функции `DuplicateHandle` и передать другому процессу, поэтому один семафор может обеспечить синхронизацию нескольких процессов. Поддерживаются также функции `up` и `down`, хотя они имеют другие названия: `ReleaseSemaphore` (для `up`) и `WaitForSingleObject` (для down). Можно определить для функции `WaitForSingleObject` предельное время простоя, и тогда вызывающий программный поток в конце концов может быть разблокирован, даже если семафор сохраняет значение 0 (однако таймеры способствуют возникновению условий гонок).

Мьютексы тоже являются объектами ядра, но они проще семафоров, поскольку у них нет счетчиков. Они, по сути, представляют собой объекты с функциями API для блокирования (`WaitForSingleObject`) и разблокирования (`ReleaseMutex`). Описатели мьютексов, как и описатели семафоров, можно дублировать и передавать другим процессам, так что программные потоки разных процессов могут иметь доступ к одному и тому же мьютексу.

Третий механизм синхронизации основан на **критических секциях**. Критические секции похожи на мьютексы, если не считать их локальности по отношению к адресному пространству исходного программного потока. Поскольку критические секции не являются объектами ядра, у них нет описателей и дескрипторов безопасности, поэтому их нельзя передавать другим процессам. Блокирование и разблокирование осуществляется с помощью функций `EnterCriticalSection` и `LeaveCriticalSection` соответственно. Так как эти функции API выполняются исключительно в пользовательском пространстве, они работают гораздо быстрее, чем мьютексы.

Последний механизм связан с использованием объектов ядра, которые называются **событиями**. Если программному потоку нужно дождаться того или иного события, он вызывает `WaitForSingleObject`. С помощью функции `SetEvent` можно разблокировать один ожидающий программный поток, а с помощью функции

`PulseEvent` — все ожидающие. Существуют несколько видов событий, которые имеют по несколько параметров.

События, мьютексы и семафоры можно определенным образом назвать и сохранить в файловой системе, как именованные каналы. Можно синхронизировать работу двух и более процессов путем открытия одного и того же события, мьютекса или семафора. В этом случае им не нужно создавать объект, а затем дублировать описатели, хотя такой подход тоже возможен.

Краткое содержание главы

Операционную систему можно считать интерпретатором некоторых возможностей архитектуры, которых нет на уровне архитектуры команд. Главными среди них являются виртуальная память, виртуальные команды ввода-вывода и поддержка параллелизма.

Виртуальная память нужна для того, чтобы позволить программам использовать больше адресного пространства, чем есть у машины на самом деле, или предоставить удобный механизм защиты и разделения памяти. Виртуальную память можно реализовать путем «чистого» разбиения на страницы, «чистой» сегментации или того и другого вместе. При страничной организации памяти адресное пространство разбивается на равные по размеру виртуальные страницы. Одни из них отображаются на физические страничные кадры, другие — нет. Обращение к отображенной странице преобразуется диспетчером памяти в правильный физический адрес. Обращение к неотображенной странице вызывает ошибку отсутствия страницы. И Core i7, и OMAP4430 имеют сложные диспетчеры памяти, поддерживающие виртуальную память и страничную организацию.

Самой важной абстракцией ввода-вывода на этом уровне является файл. Файл состоит из последовательности байтов, или логических записей, которые можно читать и записывать, не зная при этом о том, как работают диски и другие устройства ввода-вывода. Обращения к файлам могут осуществляться последовательно, произвольно по номеру записи и произвольно по ключу. Для группировки файлов используются каталоги. Файлы могут храниться в последовательных секторах, а могут быть разбросаны по всему диску. В последнем случае требуются специальные структуры данных для нахождения всех блоков файла. Чтобы отслеживать свободное пространство на диске, можно использовать список пустот (неиспользуемых областей) или битовую карту (битовое отображение).

Параллелизм часто поддерживается и реализуется в однопроцессорных системах путем разделения времени — так моделируется работа нескольких процессоров. Неконтролируемое взаимодействие различных процессов может привести к состоянию гонок. Чтобы избежать их, вводятся специальные средства синхронизации. Самыми простыми из них являются семафоры. С семафорами проблема «производитель-потребитель» решается просто и элегантно.

UNIX и Windows 7 являются сложными операционными системами. Обе системы поддерживают страничную организацию памяти и отображение файлов на память. Кроме того, они поддерживают иерархические файловые системы, где файлы состоят из последовательности байтов. Наконец, обе системы поддерживают процессы и программные потоки и предоставляют механизмы их синхронизации.

Вопросы и задания

1. Почему операционная система интерпретирует только некоторые команды уровня 3, тогда как микропрограмма интерпретирует все команды уровня архитектуры команд?
2. Машина имеет 32-разрядное виртуальное адресное пространство с побайтовой адресацией. Размер страницы составляет 4 Кбайт. Сколько существует страниц виртуального адресного пространства?
3. Должен ли размер страницы соответствовать степени двойки? Есть ли теоретическая возможность реализации страницы размером, скажем, 4000 байт? Если да, насколько такой размер оправдан?
4. Виртуальная память содержит 8 виртуальных страниц и 4 физических страничных кадра. Размер страницы составляет 1024 слова. Соответствующая таблица страниц представлена в табл. 6.10.

Таблица 6.10. Таблица страниц для задания 4

Виртуальная страница	Страничный кадр
0	3
1	1
2	Нет в основной памяти
3	Нет в основной памяти
4	2
5	Нет в основной памяти
6	0
7	Нет в основной памяти

   1) Создайте список виртуальных адресов, обращение к которым будет вызывать ошибку отсутствия страницы.
   2) Каковы физические адреса для виртуальных адресов 0, 3728, 1023, 1024, 1025, 7800 и 4096?
5. Компьютер имеет 16 страниц виртуального адресного пространства и только 4 страничных кадра. Изначально память пуста. Программа обращается к виртуальным страницам в следующем порядке:

   0, 7, 2, 7, 5, 8, 9, 2, 4

   1) Какие из обращений по алгоритму LRU вызовут ошибку отсутствия страницы?
   2) Какие из обращений по алгоритму FIFO вызовут ошибку отсутствия страницы?
6. В подразделе «Политика замещения страниц» раздела «Виртуальная память» был предложен алгоритм замещения страниц FIFO. Разработайте более эффективный алгоритм. *Подсказка*: можно обновлять счетчик во вновь загружаемой странице, оставляя все другие без изменений.

7. В системах со страничной организацией памяти, которые мы обсуждали в этой главе, обработчик ошибок отсутствия страниц был частью уровня архитектуры команд и, следовательно, отсутствовал в адресном пространстве операционной системы. На практике же такой обработчик занимает некоторые страницы и может быть удален при определенных обстоятельствах (например, в соответствии с политикой замещения страниц). Что бы случилось, если бы обработчика ошибок не было в наличии в тот момент, когда произошла ошибка? Как разрешить эту проблему?
8. Не все компьютеры содержат специальный аппаратный бит, который автоматически устанавливается при записи в страницу. Однако нужно каким-то образом следить, какие страницы изменены, чтобы не приходилось записывать все страницы обратно на диск после их использования. Если предположить, что каждая страница имеет специальные биты для разрешения чтения, записи и выполнения, то как операционная система сможет проследить, какие страницы изменялись, а какие — нет?
9. Сегментированная память содержит сегменты страниц. Каждый виртуальный адрес содержит 2-разрядный номер сегмента, 2-разрядный номер страницы и 11-разрядное смещение внутри страницы. Основная память содержит 32 Кбайт, которые разделены на страницы по 2 Кбайт. Каждый сегмент разрешается либо только читать, либо читать и выполнять, либо читать и записывать, либо читать, записывать и выполнять. Таблицы страниц и варианты защиты приведены в табл. 6.11.

Таблица 6.11. Таблицы страниц для задания 9

Сегмент 0		**Сегмент 1**		**Сегмент 2**	**Сегмент 3**	
Только чтение		**Чтение и выполнение**		**Чтение, запись и выполнение**	**Чтение и запись**	
Виртуальная страница	Страничный кадр	Виртуальная страница	Страничный кадр		Виртуальная страница	Страничный кадр
0	9	0	На диске	Таблицы страниц нет в основной памяти	0	14
1	3	1	0	Таблицы страниц нет в основной памяти	1	1
2	На диске	2	15	Таблицы страниц нет в основной памяти	2	6
3	12	3	8	Таблицы страниц нет в основной памяти	3	На диске

Вычислите физический адрес для каждого из перечисленных в табл. 6.12 вариантов доступа к виртуальной памяти. Если происходит ошибка, укажите, какого она типа.

Таблица 6.12. Варианты доступа к виртуальной памяти для задания 9

Доступ	Сегмент	Страница	Смещение внутри страницы
1. Выборка данных	0	1	1
2. Выборка данных	1	1	10
3. Выборка данных	3	3	2047
4. Сохранение данных	0	1	4
5. Сохранение данных	3	1	2
6. Сохранение данных	3	0	14
7. Переход	1	3	100
8. Выборка данных	0	2	50
9. Выборка данных	2	0	5
10. Переход	3	0	60

10. Некоторые компьютеры позволяют осуществлять ввод-вывод непосредственно в пользовательское пространство. Например, программа может начать передачу данных с диска в буфер внутри пользовательского процесса. Вызовет ли это какие-либо проблемы, если для реализации виртуальной памяти используется уплотнение? Аргументируйте.
11. Операционные системы, в которых можно использовать файлы, отображаемые на память, всегда требуют, чтобы файлы отображались в границах страниц. Например, если у нас есть 4-килобайтные страницы, файл может быть отображен, начиная с виртуального адреса 4096, а не с виртуального адреса 5000. Зачем это нужно?
12. При загрузке сегментного регистра в Core i7 вызывается соответствующий дескриптор, который загружается в невидимую часть сегментного регистра. Как вы думаете, почему проектировщики Intel выбрали такое решение?
13. Программа в компьютере Core i7 обращается к локальному сегменту 10 со смещением 8000. Поле BASE сегмента 10 в локальной таблице дескрипторов содержит число 10 000. Какой элемент таблицы страниц использует Core i7? Каков номер страницы? Каково смещение?
14. Рассмотрите возможные алгоритмы для удаления сегментов в сегментированной памяти без страничной организации.
15. Сравните внутреннюю фрагментацию с внешней. Что можно сделать, чтобы избежать каждой из них?
16. Супермаркеты часто сталкиваются с проблемой, напоминающей механизм замещением страниц в системах с виртуальной памятью. В супермаркетах есть фиксированная площадь пространства на полках, куда требуется помещать все больше и больше товаров. Если поступает новый важный продукт, например корм для собак очень высокого качества, какой-либо другой продукт нужно убрать, чтобы освободить место для нового продукта. Мы знаем два алгоритма: LRU и FIFO. Какой из них вы бы предпочли?
17. В некотором смысле технологии кэширования и страничной организации памяти похожи друг на друга. В обоих случаях память разбивается на две

области (на кэш и основную память в одном случае и на основную и дисковую память в другом). В тексте этой главы приводятся аргументы в пользу страниц как большого, так и малого размера. Справедливы ли эти аргументы в отношении размера строк кэш-памяти?

18. Почему многие файловые системы требуют, чтобы файл перед чтением явным образом открывался системным вызовом `open`?
19. Сравните применение битовой карты и списка пустот для отслеживания свободного пространства на диске. Диск состоит из 800 цилиндров, на каждом из которых 5 дорожек по 32 сектора. Сколько понадобится пустот (неиспользуемых фрагментов), чтобы их список оказался больше, чем битовая карта? Предполагается, что распределяемый блок — это сектор, а для хранения информации о каждом неиспользуемом фрагменте в списке пустот требуется 32 бита.
20. В третьей схеме распределения (после оптимальной подгонки и выбора первого подходящего варианта) пространство выделяется из самого большого оставшегося свободного блока. Какими преимуществами обладает этот алгоритм?
21. Опишите цель системного вызова открытия файла, не упомянутую в тексте.
22. Чтобы делать прогнозы относительно производительности диска, нужно иметь модель распределения памяти. Предположим, что диск рассматривается как линейное адресное пространство из N секторов ($N >> 1$). Здесь сначала идет последовательность блоков данных, затем неиспользованное пространство, затем другая последовательность блоков данных и т. д. Эмпирические измерения показывают, что вероятностные распределения длин данных и пустот одинаковы, причем для каждого из этих значений вероятность занимать i секторов составляет 2^{-i}. Каково при этом ожидаемое число пустот на диске?
23. На какой-то машине программа может создать столько файлов, сколько ей нужно, и все файлы могут увеличиваться в размерах во время выполнения программы, причем операционная система не получает никаких дополнительных данных об их конечном размере. Как вы думаете, будут ли файлы храниться в последовательных секторах? Поясните.
24. Согласно результатам исследования различных файловых систем, больше половины файлов занимают менее нескольких килобайтов дисковой памяти, причем подавляющее большинство — менее 8 Кбайт. С другой стороны, 10 % (в количественном отношении) всех файлов обычно занимает более 95 % используемого дискового пространства. Какой вывод о размере дисковых блоков можно сделать, исходя из этих данных?
25. Рассмотрим один из методов реализации команд для работы с семафорами. Всякий раз, когда центральный процессор собирается выполнить для семафора команду `up` или `down` (семафор — это целочисленная переменная в памяти), сначала он устанавливает приоритет центрального процессора таким образом, чтобы блокировать все прерывания. Затем он вызывает из памяти семафор, изменяет его и в соответствии с его значением совершает переход. После этого он снова разрешает прерывания. Будет ли этот метод работать, если:
    1) существует один центральный процессор, который переключается между процессами каждые 100 миллисекунд?
    2) два центральных процессора совместно используют общую память, в которой расположен семафор?

26. В разделе «Синхронизация процесса с использованием семафоров» сказано: «Как правило, приостановленные процессы собираются в одну очередь, что упрощает задачу их отслеживания». Какими преимуществами обладает очередь для ожидающих процессов по сравнению с активизацией случайных приостановленных процессов при выполнении `up`?
27. Компания, разрабатывающая операционные системы, получает жалобы от своих клиентов по поводу последней разработки, которая поддерживает операции с семафорами. Клиенты решили, что аморально со стороны процессов приостанавливать свою работу (то есть спать на работе). Чтобы угодить своим клиентам, компания решила добавить третью операцию, `peek`. Эта операция просто проверяет семафор, но не изменяет его и не блокирует процесс. Таким образом, программы сначала проверяют, можно ли выполнять для семафора операцию `down`. Будет ли эта идея работать, если с семафором работают три и более процессов? А если два процесса?
28. Составьте таблицу, в которой в виде функции времени от 0 до 1000 миллисекунд показано, какие из трех процессов P1, P2 и P3 работают, а какие блокированы. Все три процесса выполняют команды `up` и `down` для одного и того же семафора. Если два процесса блокированы и совершается команда `up`, то запускается процесс с меньшим номером, то есть P1 имеет преимущество над P2 и P3 и т. д. Изначально все три процесса работают, а значение семафора равно 1.
    - При t = 100 P1 выполняет операцию `down`.
    - При t = 200 P1 выполняет операцию `down`.
    - При t = 300 P2 выполняет операцию `up`.
    - При t = 400 P3 выполняет операцию `down`.
    - При t = 500 P1 выполняет операцию `down`.
    - При t = 600 P2 выполняет операцию `up`.
    - При t = 700 P2 выполняет операцию `down`.
    - При t = 800 P1 выполняет операцию `up`.
    - При t = 900 P1 выполняет операцию `up`.
29. В системе бронирования билетов на авиарейсы необходимо гарантировать, что пока один процесс использует файл, никакой другой процесс не сможет использовать этот файл. В противном случае два разных процесса, которые работают на два разных агентства по продаже билетов, смогут продать последнее оставшееся место двум пассажирам. Разработайте метод синхронизации с использованием семафоров, чтобы точно знать, что только один процесс в конкретный момент времени может получать доступ к файлу (предполагается, что процессы подчиняются определенным правилам).
30. Чтобы сделать возможной реализацию семафоров на компьютере с несколькими процессорами и общей памятью, разработчики часто включают в машину команду проверки с блокированием (назовем ее TSL — Test and Set Lock). Команда TSL X проверяет ячейку X. Если содержимое ячейки равно 0, семафоры устанавливаются на 1 за один неделимый цикл памяти, а следующая команда пропускается. Если содержимое ячейки не равно 0, TSL работает как

пустая операция. Используя команду TSL, можно написать процедуры `lock` и `unlock` со следующими свойствами: процедура `lock(x)` проверяет, заблокирована ли переменная `x`. Если нет, эта процедура блокирует переменную `x` и возвращает управление; процедура `unlock` снимет существующую блокировку. Если переменная `x` уже заблокирована, процедура просто ждет, пока она не освободится, и только после этого блокирует `x` и возвращает управление. Если все процессы блокируют таблицу семафоров перед ее использованием, то в определенный момент времени только один процесс может производить операции с переменными и указателями, что предотвращает состояние гонок. Напишите процедуры `lock` и `unlock` на ассемблере. (Вы можете делать любые разумные допущения, необходимые для решения задачи.)

31. Каковы будут значения `in` и `out` для кольцевого буфера длиной в 65 слов после каждой из следующих операций? Изначально значения `in` и `out` равны 0.
    1) 22 слова помещаются в буфер;
    2) 9 слов удаляются из буфера;
    3) 40 слов помещаются в буфер;
    4) 17 слов удаляются из буфера;
    5) 12 слов помещаются в буфер;
    6) 45 слов удаляются из буфера;
    7) 8 слов помещаются в буфер;
    8) 11 слов удаляются из буфера.
32. Предположим, что одна из версий UNIX использует блоки размером 2 Кбайт и хранит 512 дисковых адресов на каждый блок косвенной адресации (обычной косвенной адресации, двойной и тройной). Каков будет максимальный размер файла? Предполагается, что размер файловых указателей составляет 64 бита.
33. Предположим, что следующий системный вызов UNIX выполнен в контексте рис. 6.27:

    ```
    unlink("/usr/ast/bin/game3")
    ```

    Опишите подробно, какие изменения произойдут в системе каталогов.
34. Представьте, что вам нужно разработать UNIX-подобную систему для микрокомпьютера, где основной памяти недостаточно. После долгой работы системе все еще не удается уместить в память, и вы наугад выбираете какую-то системную функцию, чтобы пожертвовать ей для общего блага. Пусть этой функцией оказалась `pipe`, которая создает каналы для передачи потоков байтов от одного процесса к другому. Можно ли после этого как-то изменить ввод-вывод? Что вы можете сказать о конвейерах? Рассмотрите проблемы и возможные решения.
35. Комиссия по защите дескрипторов файлов выдвинула протест против системы UNIX, потому что при возвращении дескриптора файла она всегда возвращает самый маленький из свободных на данный момент номеров. Следовательно, дескрипторы файлов с большими номерами едва ли вообще когда-нибудь удастся использовать. Комиссия настаивает на том, чтобы система возвращала дескриптор с самым маленьким номером из тех, которые

еще не использовались программой, а не из тех, что свободны в данный момент. Комиссия утверждает, что эту идею легко реализовать, причем это не повлияет на существующие программы и, кроме того, будет гораздо справедливее по отношению к дескрипторам. А что вы думаете по этому поводу?

36. В Windows 7 можно составить список управления доступом таким образом, чтобы Светлана не имела доступа ни к одному из файлов, а все остальные имели к ним полный доступ. Как это сделать?
37. Опишите два способа программирования производителя и потребителя в Windows 7: с использованием общих буферов и семафоров. Подумайте о том, как можно реализовать общий буфер в каждом из двух случаев.
38. Работу алгоритмов замещения страниц обычно проверяют путем моделирования. Предположим, что вам нужно написать моделирующую программу, реализующую виртуальную память со страничной организацией для машины, содержащей 64 страницы по 1 Кбайт. Программа должна поддерживать одну таблицу из 64 элементов, по одному элементу на страницу. Каждый элемент таблицы содержит номер физической страницы, который соответствует данной виртуальной странице. Моделирующая программа должна считывать файл, содержащий виртуальные адреса в десятичной системе счисления, по одному адресу на строку. Если соответствующая страница находится в памяти, просто фиксируйте факт наличия страницы. Если ее нет в памяти, вызовите процедуру замещения страниц, чтобы выбрать страницу, которую можно удалить (то есть переписать элемент таблицы), и зафиксируйте факт отсутствия страницы. Никакой передачи страниц реализовывать не нужно. Создайте файл, состоящий из непоследовательных адресов, и проверьте производительность двух алгоритмов: LRU и FIFO. А теперь создайте файл адресов, в котором x % адресов находятся на 4 байта выше, чем предыдущие. Проведите тесты для различных значений x и сообщите о полученных результатах.
39. В программе, показанной в листинге 6.1, имеет место фатальная гонка, так как два программных потока бесконтрольно обращаются к общим переменным. При этом не задействуются ни семафоры, ни какие-либо другие методики взаимного исключения. Запустите эту программу, и вы увидите, что через некоторое время она зависнет. Если она все же не зависнет, попробуйте сделать код более уязвимым, разместив между операторами корректировки значений `m.in` и `m.out` и операторами их проверки какие-нибудь вычисления. Какой объем вычислений нужно ввести в эту программу, чтобы она зависала, скажем, раз в час?
40. Напишите программу для UNIX или Windows 7, которая на входе получает имя каталога. Программа должна печатать список файлов этого каталога, каждый файл на отдельной строке, а после имени файла — его размер. Имена файлов должны располагаться в том порядке, в котором они располагаются в каталоге. Неиспользованные позиции каталога должны выводиться с пометкой (не использован).

Глава 7
Уровень ассемблера

В главах 4–6 мы обсуждали три уровня, которые можно встретить в архитектуре большинства современных компьютеров. В этой главе речь пойдет о еще одном уровне, который также присутствует в архитектуре практически всех современных машин. Это — уровень ассемблера. Этот уровень существенно отличается от трех предыдущих, поскольку он реализуется путем трансляции, а не интерпретации.

Для перевода пользовательских программ с одного языка программирования на другой разработаны специальные программы, которые называются **трансляторами**. Язык, на котором изначально была написана программа, называется **входным**, или **исходным**, **языком**, а язык, на который она транслируется, — **выходным**, или **целевым**. И входной, и выходной языки определяют содержание уровней иерархии. Если имеется процессор, который может выполнять программы, написанные на входном языке, то нет необходимости транслировать исходную программу на другой язык.

Трансляция необходима в том случае, если есть процессор (реализованный аппаратно или программно) для выходного языка, но нет процессора для входного языка. Если трансляция выполнена правильно, то оттранслированная программа будет давать точно такие же результаты, что и исходная (если бы существовал подходящий для нее процессор). Следовательно, имеется возможность организовать новый уровень, который сначала будет транслировать программы, написанные для выходного уровня, а затем сам их выполнять.

Важно понимать разницу между трансляцией и интерпретацией[1]. При трансляции исходная программа на входном языке сразу не выполняется. Сначала она преобразуется в эквивалентную программу, так называемую **объектную**, или **исполняемую двоичную**, **программу**, которая выполняется только после завершения трансляции. То есть при трансляции нужно пройти два шага:

1. Создание эквивалентной программы на выходном языке.
2. Выполнение полученной программы.

Эти два шага выполняются не одновременно. Второй шаг начинается только после завершения первого. В интерпретации есть только один шаг: выполнение исходной программы. Никакого эквивалентного кода генерировать не нужно, хотя иногда для упрощения интерпретации исходная программа преобразуется в промежуточную форму (например, в Java-код).

Во время выполнения объектной программы задействованы только три уровня: микроархитектуры, архитектуры команд и операционной системы. Следовательно, во время работы программы в памяти компьютера находятся

[1] В отечественной литературе принято и интерпретацию, и компиляцию называть трансляцией (именно компиляцию автор здесь называет трансляцией). Другими словами, трансляторы могут быть либо компиляторами, либо интерпретаторами. — *Примеч. науч. ред.*

три программы: пользовательская объектная программа, операционная система и микропрограмма (если она есть). Никаких следов исходной программы не остается. То есть число уровней при выполнении программы может не соответствовать числу уровней до трансляции. Следует отметить, в данной книге мы определяем принадлежность к тому или иному уровню по командам и языковым конструкциям, доступным программистам этого уровня (а не по технологии реализации), в то время как некоторые авторы проводят различие между уровнями, реализованными интерпретаторами и трансляторами.

Знакомство с ассемблером

Трансляторы можно условно разделить на две группы в зависимости от отношения между входным и выходным языками. Если входной язык является символическим представлением числового машинного языка, то транслятор называется **ассемблером**, а входной язык — **языком ассемблера**, или просто **ассемблером**. Если входной язык является языком высокого уровня (например, Java или C), а выходной язык является либо числовым машинным языком, либо символическим представлением последнего, то транслятор называется **компилятором**.

Что такое «язык ассемблера»?

Язык ассемблера — это язык, в котором каждый оператор соответствует ровно одной машинной команде. Иными словами, в программе, написанной на ассемблере, существует взаимно однозначное соответствие между машинными командами и операторами. Если каждая строка в ассемблерной программе содержит ровно один оператор и каждое машинное слово содержит ровно одну команду, то из ассемблерной программы размером в n строк получится программа на машинном языке из n слов.

Мы программируем на языке ассемблера, а не на машинном языке (в шестнадцатеричной системе счисления), поскольку это гораздо проще. Использовать символические имена и адреса вместо двоичных и восьмеричных намного удобнее. Многие могут запомнить, что обозначениями для сложения (add), вычитания (subtract), умножения (multiply) и деления (divide) служат команды `ADD`, `SUB`, `MUL` и `DIV`, но мало кто сможет запомнить соответствующие числовые коды, которые использует для этих команд машина. Программисту, пишущему на языке ассемблера, нужно знать только символические названия, поскольку ассемблер транслирует их в машинные команды.

Это утверждение касается и адресов. Программист, пишущий на языке ассемблера, может давать символические имена ячейкам памяти, и уже ассемблер должен позаботиться о том, чтобы получить из них правильные числовые значения. В то же время программисту, пишущему на машинном языке, всегда приходится работать с числовыми значениями адресов. Сейчас уже нет программистов, пишущих программы на машинном языке, хотя несколько десятилетий назад до изобретения ассемблеров программы именно так и писались.

Язык ассемблера имеет несколько особенностей, отличающих его от языков высокого уровня. Во-первых, это взаимно однозначное соответствие между опе-

раторами языка ассемблера и машинными командами (об этом мы уже говорили). Во-вторых, программисту, пишущему на ассемблере, доступны все объекты и команды целевой машины. Программист, пишущий на языках высокого уровня, такой свободы не имеет. Например, если целевая машина содержит бит переполнения, ассемблерная программа может проверить его, а Java-программа — нет. Ассемблерная программа может выполнить любую команду из набора команд целевой машины, а программа на языке высокого уровня — нет. Короче говоря, все, что можно сделать на машинном языке, можно сделать и на ассемблере, но в то же время программистам, пишущим программы на языках высокого уровня, недоступны многие команды, регистры и другие объекты. Языки системного программирования (например, C) часто занимают промежуточное положение. Они хотя и обладают синтаксисом, присущим языкам высокого уровня, с точки зрения возможностей доступа ближе к ассемблеру.

Наконец, ассемблерная программа может работать только на компьютерах одного семейства, а программа, написанная на языке высокого уровня, теоретически может работать на разных машинах. Возможность переноса программного обеспечения с одной машины на другую очень важна для многих прикладных программ.

Назначение ассемблера

Работать с языком ассемблера не просто. Написание одной и той же программы на ассемблере занимает гораздо больше времени, чем на языке высокого уровня. Кроме того, очень много времени занимает отладка, а сопровождение кода требует гораздо больших усилий.

Но зачем же тогда вообще писать программы на ассемблере? Есть две причины: производительность и доступ к аппаратуре. Прежде всего, квалифицированный программист, пишущий на ассемблере, может составить гораздо меньшую по объему и гораздо более быстродействующую программу, чем программа, написанная на языке высокого уровня. Для некоторых программ быстродействие и объем имеют чрезвычайно важное значение. К этой категории относятся многие встроенные прикладные программы (например, в смарт-картах, сотовых телефонах, драйверах устройств) и процедуры BIOS, а также внутренние циклы приложений, критичных по времени.

Следующая причина заключается в том, что некоторым процедурам требуется полный доступ к аппаратному обеспечению, который обычно невозможно обеспечить средствами языков высокого уровня. В эту категорию попадают обработчики прерываний и исключений операционных систем, а также контроллеры устройств встроенных систем, работающих в режиме реального времени.

Кроме этих двух причин существует еще две. Во-первых, компилятор должен либо на выходе производить программу, которая может использоваться ассемблером, либо самостоятельно выполнять ассемблирование. Таким образом, знание языка ассемблера существенно для понимания того, как работает компилятор. В конце концов, кто-то ведь должен написать компилятор (и его ассемблер).

Во-вторых, ассемблер дает прекрасное представление о реальной машине. Для тех кто изучает архитектуру компьютеров, написание ассемблерного кода — единственный способ узнать, что собой представляет машина на уровне архитектуры.

Формат операторов в ассемблере

Хотя структура ассемблерного оператора отражает структуру соответствующей машинной команды, языки ассемблера для разных машин и разных уровней во многом похожи, что позволяет говорить о языке ассемблера в целом. В листингах 7.1–7.3 приведен фрагмент программы на ассемблере x86, вычисляющей формулу `N = I + J`. Операторы под пустой строкой резервируют память для переменных `I`, `J` и `N`, то есть не являются символьными представлениями машинных команд.

Листинг 7.1. Вычисление выражения N = I+ Jна ассемблере x86

```
FORMULA:  MOV     EAX,I           ; регистр EAX = I
          ADD     EAX,J           ; регистр EAX = I + J
          MOV     N,EAX           ; N = I + J

I         DD      3               ; резервирование 4 байт
                                  ; и их инициализация значением 3
J         DD      4               ; резервирование 4 байт
                                  ; и их инициализация значением 4
N         DD      0               ; резервирование 4 байт
                                  ; и их инициализация значением 0
```

Для компьютеров семейства Intel (то есть x86) существуют несколько ассемблеров, которые отличаются друг от друга синтаксисом. В этой главе мы будем использовать язык ассемблера Microsoft MASM. Также существует немало ассемблеров для процессоров ARM, но по своему синтаксису они близки к ассемблеру x86, поэтому одного примера должно быть достаточно.

Ассемблерные операторы состоят из четырех полей: метки, операции, операндов и комментариев. Метки служат символическими именами для адресов памяти. Они позволяют переходить к командам и данным, позволяя по символическому имени получить доступ к тому месту, где хранятся команды и данные. Если оператор снабжен меткой, то эта метка обычно располагается в начале строки.

В листинге примера присутствуют метки: FORMULA, I, J и N. В ассемблере MASM двоеточие ставится только после меток команд, но не после меток данных. Данное различие вовсе не является чем-то фундаментальным, просто у разработчиков разных ассемблеров разные вкусы. Архитектура машины никак не влияет на тот или иной выбор. Единственное достоинство двоеточия состоит в том, что метку можно написать на отдельной строке, а код операции — на следующей строке с тем же отступом, что и метка. Без двоеточия компилятор не мог бы отличить метку от кода операции при их размещении в отдельных строках.

В некоторых ассемблерах длина метки ограничена значением 6 или 8 символов. В то же время в большинстве языков высокого уровня длина имен произвольна. Длинные и хорошо подобранные имена упрощают чтение и понимание программы.

В каждой машине есть несколько регистров, но названия у них совершенно разные. Регистры Core i7 называются EAX, EBX, ECX и т. д.

В поле кода операции содержится либо символическая аббревиатура этого кода (если оператор является символическим представлением машинной команды), либо директива для ассемблера. Выбор имени — дело вкуса, и поэтому

разные разработчики называют их по-разному. Разработчики ассемблера MASM решили использовать обозначение MOV и для загрузки регистра из памяти, и сохранения регистра в память. С таким же успехом они могли использовать MOVE в паре с LOAD или STORE.

Программам на языке ассемблера часто требуется резервировать пространство для данных. Разработчики MASM выбрали для этой операции название DD (Define Double — определить двойное слово), поскольку слово процессора 8088 имело длину 16 бит.

В поле операндов оператора задаются адреса и регистры, которые являются операндами машинной команды. В поле операндов команды целочисленного сложения указывается, что и к чему нужно прибавить. Поле операндов команд перехода определяет, куда совершается переход. Операндами могут быть регистры, константы, ячейки памяти и т. д.

В поле комментариев программист размещает свои пояснения, касающиеся работы программы. Эти пояснения могут пригодиться программистам, которым потом придется использовать и дорабатывать чужую программу, а также самому автору программы, когда он через год вернется к работе над ней. Ассемблерная программа без таких комментариев — нечто совершенно невразумительное (даже для ее автора). Комментарии могут быть полезны только людям и никак не влияют на работу программы.

Директивы

Ассемблерная программа определяет не только машинные команды, которые нужно выполнять процессору, но и команды, которые нужно выполнять самому ассемблеру (например, выделить немного памяти или выдать новую страницу листинга). Команды для ассемблера называются **псевдокомандами**, или **ассемблерными директивами**. В листинге 7.1 мы уже встречали типичную псевдокоманду DD. В табл. 7.1 перечислены некоторые другие псевдокоманды (директивы) ассемблера MASM для платформы x86.

Таблица 7.1. Некоторые директивы ассемблера MASM

Директива	Описание
SEGMENT	Начало нового сегмента (текста, данных и т. п.) с определенными атрибутами
ENDS	Завершение текущего сегмента
ALIGN	Управление выравниванием следующей команды или данных
EQU	Определение нового символа, равного данному выражению
DB	Выделение памяти для одного или нескольких байтов
DW	Выделение памяти для одного или нескольких 16-разрядных полуслов
DD	Выделение памяти для одного или нескольких 32-разрядных слов
DQ	Выделение памяти для одного или нескольких 64-разрядных двойных слов
PROC	Начало процедуры

продолжение ⇩

Таблица 7.1 *(продолжение)*

Директива	Описание
ENDP	Завершение процедуры
MACRO	Начало макроса
ENDM	Завершение макроса
PUBLIC	Экспорт имени, определенного в данном модуле
EXTERN	Импорт имени из другого модуля
INCLUDE	Вызов другого файла и включение его в текущий файл
IF	Начало условного ассемблирования программы на основе данного выражения
ELSE	Начало условного ассемблирования программы, если условие для директивы IF не выполнено
ENDIF	Завершение условного ассемблирования программы
COMMENT	Определение нового символа начала поля комментариев
PAGE	Принудительный разрыв страницы в листинге
END	Завершение ассемблерной программы

Директива `SEGMENT` начинает новый сегмент, а директива `ENDS` завершает его. Разрешается начинать текстовый сегмент, затем начинать сегмент данных, затем переходить обратно к текстовому сегменту и т. д.

Директива `ALIGN` передает следующую строку (обычно данные) по адресу, заданному аргументом директивы. Например, если текущий сегмент уже содержит 61 байт данных, тогда после выполнения директивы `ALIGN 4` следующим выделяемым адресом будет адрес 64.

Директива `EQU` дает символическое название некоторому выражению. Например, после следующей директивы символ `BASE` можно использовать в программе вместо значения 1000:

```
BASE EQU 1000
```

Выражение, которое следует за директивой `EQU`, может содержать несколько символов, соединенных знаками арифметических и других операций, например:

```
LIMIT EQU 4 * BASE + 2000
```

Большинство ассемблеров, в том числе MASM, требуют, чтобы символ был определен в программе до его появления в таком выражении, как это.

Директивы `DB`, `DD`, `DW` и `DQ` выделяют память для одной или нескольких переменных размером 1, 2, 4 и 8 байт соответственно. Например:

```
TABLE DB 11, 23, 49
```

Эта директива выделяет место для 3 байт и присваивает им начальные значения 11, 23 и 49 соответственно, кроме того, она определяет символ `TABLE`, равный тому адресу, по которому хранится значение 11.

Директивы `PROC` и `ENDP` определяют начало и конец ассемблерных процедур. Процедуры в ассемблере исполняют ту же роль, что и в языках программирования высокого уровня. Директивы `MACRO` и `ENDM` определяют начало и конец макроса. О макросах мы поговорим в следующем разделе.

Директивы PUBLIC и EXTERN управляют видимостью (доступность) символических имен. Программы часто пишут в виде совокупности файлов. Иногда процедуре, находящейся в одном файле, нужно вызвать процедуру или получить доступ к данным, определенным в другом файле. Чтобы такие перекрестные ссылки между файлами стали возможными, символ (имя), который нужно сделать доступным для других файлов, экспортируется с помощью директивы PUBLIC. Чтобы ассемблер не выдавал предупреждений по поводу использования символа, который не определен в данном файле, этот символ может быть объявлен внешним (EXTERN), то есть определенным в другом файле. Символы, которые не определены ни в одной из этих директив, могут использоваться только в пределах одного файла. Поэтому даже если, например, символ FOO встречается в нескольких файлах, это не вызовет никакого конфликта, поскольку указанный символ локален по отношению к каждому файлу.

Директива INCLUDE заставляет ассемблер вызвать другой файл и включить его в текущий. Такие включенные файлы часто содержат определения, макросы и другие элементы, необходимые для разных файлов.

Многие языки ассемблера, в том числе MASM, поддерживают условное ассемблирование программы. Например:

```
WORDSIZE EQU 32
IF WORDSIZE GT 32
WSIZE: DD 64
ELSE
WSIZE: DD 32
ENDIF
```

Эта программа выделяет в памяти одно 32-разрядное слово и вызывает его адрес WSIZE. Этому слову придается одно из значений: либо 32, либо 16 в зависимости от значения WORDSIZE (в данном случае — 16). Такая конструкция может использоваться в программе, которая может ассемблироваться как в 32-, так и в 64-разрядном режиме. Если в начале и в конце машинно зависимого кода вставить директивы IF и ENDIF, а затем изменить единственное определение, WORDSIZE, программу можно автоматически настроить на один из двух размеров. Применяя такой подход, можно задействовать одну такую исходную программу для нескольких разных машин. В большинстве случаев все машинно зависимые определения, такие как WORDSIZE, сохраняются в одном файле, причем для разных машин должны быть разные файлы. Путем включения файла с нужными определениями программу можно легко перекомпилировать для разных машин.

Директива COMMENT позволяет пользователю заменить символ начала комментария (точку с запятой) чем-либо другим. Директива PAGE используется для управления листингом программы. Наконец, директивой END помечается конец программы.

В ассемблере MASM есть еще много директив. Другие ассемблеры x86 содержат другой набор директив, поскольку эти директивы определяются не архитектурой машины, а вкусами разработчиков ассемблера.

Макросы

Обычно программистам, пишущим на языке ассемблера, приходится многократно повторять одни и те же цепочки команд. Хотя проще всего писать нужные

команды всякий раз, когда они требуются, это занятие становится утомительным, особенно если последовательность достаточно длинная или если ее нужно повторять слишком часто.

Конечно, можно оформить эту последовательность в процедуру и вызывать ее в случае необходимости. Однако у такой стратегии тоже есть свои недостатки, поскольку в этом случае каждый раз придется выполнять специальную команду вызова процедуры и команду возврата. Если последовательности команд короткие (например, всего две команды), но используются часто, то вызовы процедур могут значительно сказаться на быстродействии программы. Простым и эффективным решением этой проблемы являются макросы.

Макроопределение, макровызов и макрорасширение

Макроопределение — это способ дать имя фрагменту кода. После того как макрос определен, программист может вместо фрагмента кода писать имя макроса. В сущности, макрос — это просто имя фрагмента кода. В листинге 7.2 приведена ассемблерная программа для x86, которая дважды меняет местами значения переменных `P` и `Q`. Вот как выглядит основная цепочка операторов:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

В листинге 7.3 эта последовательность определяется как макрос `SWAP`. После определения макроса каждое вхождение его имени заменяется четырьмя строками кода.

Листинг 7.2. Смена значений переменных P и Q без использования макроса

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX

MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

Листинг 7.3. Смена значений переменных P и Q с использованием макроса

```
SWAP  MACRO
      MOV EAX,P
      MOV EBX,Q
      MOV Q,EAX
      MOV P,EBX
      ENDM

      SWAP

      SWAP
```

Хотя в разных языках ассемблера определение макроса выглядит немного по-разному, во всех оно состоит из одних и тех же базовых частей:

- заголовка макроса, в котором дается имя определяемого макроса;
- текст, в котором приводится тело макроса;
- директивы, которая завершает определение (например, `ENDM`).

Когда ассемблер наталкивается на макроопределение в программе, он сохраняет его в таблице макроопределений для последующего использования. Всякий раз, когда в программе в качестве кода операции будет появляться макрос (в нашем примере — SWAP), ассемблер заменит его телом макроса. Использование имени макроса в качестве кода операции называется **макровызовом**, а его замена телом макроса — **макрорасширением**.

Макрорасширение происходит в ходе ассемблирования, а не во время выполнения программы. Этот момент очень важен. Программы, приведенные в листингах 7.2 и 7.3, порождают один и тот же машинный код. По программе на машинном языке невозможно определить, использовались макросы при ее порождении или нет. В полученной программе никаких признаков макросов не остается.

Макровызовы не следует путать с вызовами процедур. Основное отличие состоит в том, что макровызов — это команда ассемблеру заменить имя макроса телом макроса. Вызов процедуры — это машинная команда, которая, будучи вставлена в объектную программу, позднее должна быть выполнена для вызова процедуры. В табл. 7.2 сравниваются макровызовы и вызовы процедур.

Таблица 7.2. Сравнение макровызовов и вызовов процедур

Вопрос	Макровызов	Вызов процедуры
Когда совершается вызов?	Во время ассемблирования	Во время выполнения программы
Вставляется ли тело макроса или процедуры в объектную программу каждый раз, когда совершается вызов?	Да	Нет
Вставляется ли в объектную программу, команда вызова процедуры, которая затем выполняется?	Нет	Да
Нужно ли после вызова использовать команду возврата?	Нет	Да
Сколько копий тела макровызова или процедуры появляется в объектной программе?	Одна на каждый макровызов	Одна

На концептуальном уровне можно считать, что процесс ассемблирования проходит в два прохода. На первом проходе сохраняются все макроопределения, а макровызовы расширяются. На втором проходе обрабатывается полученный в результате текст. Иными словами, исходная программа считывается, а затем трансформируется в другую программу, из которой удалены все макроопределения и в которой каждый макровызов заменен телом макроса. Полученная программа без макросов затем передается ассемблеру.

Важно иметь в виду, что программа представляет собой строку символов, каковыми могут быть буквы, цифры, пробелы, знаки пунктуации и символы

возврата каретки (перехода на новую строку). При макрорасширении определенные подстроки из этой строки заменяются другими символьными строками. Макросы — средство манипулирования символьными строками без изменения их значения.

Макросы с параметрами

Описанные в предыдущем подразделе макросы можно использовать для сокращения объемы программ, в которых часто повторяется одна и та же последовательность команд. Однако иногда программа содержит несколько похожих, но не идентичных последовательностей команд. Например, в листинге 7.4 первая последовательность меняет местами значения переменных `P` и `Q`, а вторая — переменных `R` и `S`.

Листинг 7.4. Смена значений двух пар переменных без использования макроса

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX

MOV EAX,R
MOV EBX,S
MOV S,EAX
MOV R,EBX
```

Для работы с такими почти идентичными последовательностями предусмотрены макроопределения, предлагающие **формальные параметры**, и макровызовы, в которых формальные параметры заменяются **фактическими параметрами**. Фактические параметры помещаются в поле операндов макровызова. В листинге 7.5 представлена программа из листинга 7.4, в которую включен макрос с двумя параметрами. Символические имена `P1` и `P2` — это формальные параметры. Во время расширения макроса каждое вхождение `P1` внутри тела макроса заменяется первым фактическим параметром, а каждое вхождение `P2` — вторым фактическим параметром. Пример:

```
CHANGE P,Q
```

В этом макровызове `P` — это первый фактический параметр, а `Q` — второй фактический параметр. Таким образом, программы в листингах 7.4 и 7.5 идентичны.

Листинг 7.5. Смена значений двух пар переменных с использованием макроса

```
CHANGE  MACRO P1,P2
        MOV EAX,P1
        MOV EBX,P2
        MOV P2,EAX
        MOV P1,EBX
        ENDM

        CHANGE P,Q

        CHANGE R,S
```

Дополнительные возможности

Большинство макропроцессоров поддерживают целый ряд дополнительных функций, которые упрощают работу программиста, пишущего на языке ассемблера. В этом подразделе мы рассмотрим несколько дополнительных функций ассемблера MASM. Для всех ассемблеров характерна проблема: дублирование меток. Предположим, что макрос содержит команду условного перехода и метку, к которой совершается переход. Если макрос вызывается два и более раз, метка будет дублироваться, что вызовет ошибку. Поэтому программист должен приписывать каждому вызову в качестве параметра свою метку. Другое решение (оно применяется в MASM) — объявлять метку локальной (`LOCAL`), при этом ассемблер автоматически будет порождать другую метку при каждом расширении макроса. В некоторых ассемблерах числовые метки по умолчанию считаются локальными.

MASM и большинство других ассемблеров позволяют определять макросы внутри других макросов. Эта особенность очень полезна в сочетании с условным ассемблированием программы. Обычно один и тот же макрос определяется в обеих ветвях оператора `IF`:

```
M1  MACRO
    IF WORDSIZE GT 16
M2    MACRO
      ...
      ENDM
    ELSE
M2    MACRO
    ...
      ENDM
    ENDIF
    ENDM
```

В любом случае макрос `M2` будет определен, но определение зависит от того, на какой машине ассемблируется программа: на 16-разрядной или на 32-разрядной. Если `M1` не вызывается, макрос `M2` вообще не будет определен.

Одни макросы могут вызывать другие макросы, в том числе самих себя. Если макрос рекурсивный, то есть вызывает самого себя, он должен передавать самому себе параметр (который изменяется при каждом расширении), а затем проверять этот параметр и завершать рекурсию, когда параметр достигает определенного значения. В противном случае получится бесконечный цикл, и тогда пользователю придется завершить ассемблирование вручную.

Реализация макросов в ассемблере

Для реализации макросов ассемблер должен уметь выполнять две функции: сохранять макроопределения и расширять макровызовы. Мы рассмотрим эти функции по очереди.

Ассемблер должен сохранять таблицу всех имен макросов, в которой каждое имя сопровождается указателем на определение этого макроса, чтобы его можно было получить в случае необходимости. В одних ассемблерах предусмотрена отдельная таблица для имен макросов, а другие содержат общую таблицу,

в которой находятся не только имена макросов, но и все машинные команды и директивы.

При обнаружении макроопределения создается новый элемент таблицы с именем макроса, числом параметров и указателем на другую таблицу, в которой будет храниться тело макроса. В этот момент также создается список формальных параметров. Затем считывается тело макроса и сохраняется в таблице макроопределений. Формальные параметры, которые встречаются в теле цикла, обозначаются специальным символом. Ниже приведен пример внутреннего представления макроса `CHANGE`. Для обозначения символа возврата каретки используется точка с запятой, а для обозначения формального параметра — символ `&`.

```
MOV EAX,&P1;MOV EBX,&P2;MOV &P2EAX;MOV &P1,EBX;
```

В таблице макроопределений тело макроса представляет собой просто символьную строку.

При ассемблировании во время первого прохода отыскиваются коды операций, а макросы расширяются. Всякий раз, когда встречается макроопределение, оно сохраняется в таблице макросов. При вызове макроса ассемблер временно приостанавливает чтение входных данных из входного устройства и начинает считывать сохраненное тело макроса. Формальные параметры, извлеченные из тела макроса, заменяются фактическими параметрами, которые предоставляются при вызове. Символ `&` перед параметрами помогает ассемблеру опознавать их.

Процесс ассемблирования

В следующих подразделах показано, как работает ассемблер. И хотя ассемблеры разных машин разные, процесс ассемблирования, по сути, один и тот же.

Ассемблирование за два прохода

Поскольку ассемблерная программа состоит из ряда операторов, на первый взгляд может показаться, что ассемблер сначала должен считать оператор, затем перевести его на машинный язык и, наконец, передать полученный машинный язык в файл, а соответствующий фрагмент листинга — в другой файл. Этот процесс должен повторяться до тех пор, пока вся программа не будет оттранслирована. Однако, к сожалению, такая стратегия не работает.

Рассмотрим ситуацию, где первый оператор — переход к адресу `L`. Ассемблер не может ассемблировать оператор, пока не знает адрес `L`. Но адрес `L` может находиться где-нибудь в конце программы, и тогда ассемблер не сможет найти этот адрес, не прочитав всю программу. Эта проблема называется **проблемой опережающей ссылки** (forward reference problem) и заключается она в том, что символическое имя `L` используется еще до своего определения (то есть выполняется обращение к символическому имени, определение которого появится позднее).

Опережающие ссылки можно обрабатывать двумя способами. Во-первых, ассемблер может прочитать программу дважды. Каждое прочтение исходной программы называется **проходом**, а транслятор, который читает исходную программу дважды, называется **двухпроходным**. На первом проходе собираются и со-

храняются в таблице все определения символических имен, в том числе метки. К началу второго прохода значения символических имен уже известны, никаких опережающих ссылок нет, и каждый оператор можно читать и ассемблировать. Хотя при этом требуется дополнительный проход по исходной программе, зато такая стратегия относительно проста.

При втором подходе ассемблерная программа читается один раз и преобразуется в промежуточную форму, и эта промежуточная форма сохраняется в таблице. Затем совершает второй проход, но уже не по исходной программе, а по таблице. Если физической (или виртуальной) памяти для этого подхода достаточно, экономится время, затрачиваемое на процесс ввода-вывода. Если при ассемблировании требуется вывести листинг, тогда полностью сохраняются исходные операторы, включая комментарии. Если листинг не нужен, то промежуточную форму можно сократить, оставив только самое существенное.

Еще одна цель первого прохода — сохранять все макроопределения и расширять вызовы по мере их появления. Следовательно, в одном проходе происходит и определение символических имен, и расширение макросов.

Первый проход

Главная цель первого прохода — построить **таблицу символических имен**, содержащую значения всех символических имен, которым с помощью директивы приписывается определенная метка или значение:

```
BUFSIZE EQU 8192
```

Приписывая значение символическому имени в поле метки команды, ассемблер должен знать, какой адрес будет иметь эта команда во время выполнения программы. Для этого ассемблер во время ассемблирования сохраняет специальную переменную, называемую **счетчиком адресов команд** (Instruction Location Counter, **ILC**). В начале первого прохода эта переменная устанавливается в 0 и увеличивается после каждой обработанной команды на длину этой команды. В листинге 7.6 дан пример для x86.

Листинг 7.6. Счетчик адресов команд позволяет отслеживать адреса команд

```
MARIA:    MOV  EAX, I      ; EAX = I                        5        100
          MOV  EBX, J      ; EBX = J                        6        105
ROBERTA:  MOV  ECX, K      ; ECX = K                        6        111
          IMUL EAX, EAX    ; EAX = I * I                    2        117
          IMUL EBX, EBX    ; EBX = J * J                    3        119
          IMUL ECX, ECX    ; ECX = K * K                    3        122
MARILYN:  ADD  EAX, EBX    ; EAX = I * I + J * J            2        125
          ADD  EAX, ECX    ; EAX = I * I + J * J + K * K    2        127
STEPHANY: JMP  DONE        ; переход к DONE                 5        129
```

При первом проходе в большинстве ассемблеров используются, по крайней мере, три таблицы: таблица символических имен, таблица директив и таблица кодов операций. В случае необходимости используется еще литеральная таблица. Таблица символических имен содержит по одной записи для каждого имени, как показано в табл. 7.3. Символические имена либо являются метками, либо явным образом определяются (например, с помощью директивы `EQU`). В каждом элементе таблицы символических имен содержится само имя (или указатель на

него), его численное значение и иногда некоторая дополнительная информация. Она может включать:

- длину поля данных, связанного с символическим именем;
- биты перераспределения памяти (которые показывают, изменится ли значение символа, если программа будет загружена не по тому адресу, по которому ее предполагал загрузить ассемблер);
- сведения о том, можно ли получить доступ к символическому имени извне процедуры.

Таблица 7.3. Таблица символических имен для программы из листинга 7.8

Символическое имя	Значение	Прочая информация
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

В таблице кодов операций предусмотрена, по крайней мере, одна запись для каждого символьного кода операции ассемблера (табл. 7.4). В каждой записи содержится символьный код операции, два операнда, числовое значение кода операции, длина команды и номер типа, по которому можно определить, к какой группе относится код операции (коды операций делятся на группы в зависимости от числа и типа операндов).

Таблица 7.4. Несколько элементов таблицы кодов операций ассемблера x86

Код операции	Первый операнд	Второй операнд	Шестнадцатеричный код	Длина команды	Класс команды
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

В качестве примера рассмотрим код операции `ADD`. Если первым операндом команды `ADD` является регистр EAX, а вторым — 32-разрядная константа (`immed32`), то используется код операции 0x05, а длина команды составляет 5 байт (для констант, которые могут быть выражены 8 и 16 битами, используются другие коды, не приведенные в таблице). Если оба операнда команды `ADD` являются регистрами, длина команды составляет 2 байта, а код операции равен 0x01. Все комбинации кодов операций и операндов, которые соответствуют данному правилу, относятся к классу 19 и обрабатываются так же, как команда `ADD` с двумя регистрами в качестве операндов. Класс команд идентифицирует процедуру, которая вызывается для обработки всех команд данного типа.

В некоторых ассемблерах можно писать команды с применением непосредственной адресации, даже если соответствующей команды нет в выходном языке. Такие команды с «псевдонепосредственными» адресами обрабатываются

следующим образом. Ассемблер назначает область памяти для непосредственного операнда в конце программы и порождает команду, которая к нему обращается. Например, универсальная вычислительная машина IBM 3090 не имеет команд с непосредственными адресами. Тем не менее для загрузки в регистр 14 константы 5 размером в полное слово программист может написать команду:

```
L 14,=F'5'
```

Таким образом, программисту не нужно писать директиву, чтобы разместить слово в памяти, присвоить ему значение 5, дать ему метку, а затем использовать эту метку в команде `L`. Константы, для которых ассемблер автоматически резервирует память, называются **литералами**. Литералы упрощают чтение и понимание программы; с ними значение константы становится очевидным уже при чтении в исходного оператора. При первом проходе ассемблер строит всех литералов, которые используются в программе. Все три компьютера, которые мы взяли в качестве примеров, имеют команды с непосредственными адресами, поэтому их ассемблеры не поддерживают литералов. Команды с непосредственными адресами в настоящее время считаются вполне обычными, хотя раньше они рассматривались как нечто совершенно экзотическое. Вероятно, популярность литералов внушила разработчикам, что непосредственная адресация — очень хорошая идея. Если литералы нужны, то во время ассемблирования сохраняется таблица литералов, в которой появляется новый элемент всякий раз, когда встречается литерал. После первого прохода таблица сортируются, и повторяющиеся элементы удаляются.

В листинге 7.7 приведена процедура, которая может быть заложена в основу первого прохода ассемблера. Названия команд выбраны таким образом, чтобы была ясна их суть. Этот листинг представляет собой хорошую отправную точку для изучения ассемблера. Он достаточно короткий, понятный и из него видно, каким должен быть следующий шаг — это написание процедур, которые упоминаются в данном листинге.

Листинг 7.7. Первый проход простого ассемблера

```
public static void pass_one() {
// Схема первого прохода ассемблера
  boolean more_input=true;      // флаг, который останавливает первый проход
  String line, symbol, literal, opcode;      // поля команды
  int location_counter, length, value, type; // переменные
  final int END_STATEMENT = -2; // сигналы окончания ввода

  location_counter = 0;         // ассемблирование первой команды в ячейке 0
  initialize_tables();          // общая инициализация

while (more_input) {            // more_input с помощью директивы END
                                // получает значение "ложь"
    line = read_next_line();    // считывание строки
    length = 0;                 // # байт в команде
    type = 0;                   // тип команды

    if (line_is_not_comment(line)) {
      symbol = check_for_symbol(line);   // содержит ли строка метку?
      if (symbol != null)                // если да, то записываются
                                         // символ и значение
```

продолжение ⇩

Листинг 7.7 *(продолжение)*

```
        enter_new_symbol(symbol, location_counter);
      literal = check_for_literal(line);  // содержит ли строка литерал?
      if (literal != null)                // если да, то он
                                          // сохраняется в таблице
        enter_new_literal(literal);
// Теперь определяем тип кода операции. -1 значит недопустимый код операции
      opcode = extract_opcode(line);      // определяем место кода операции
      type =search_opcode_table(opcode);  // находим формат,
                                          // например, OP REG1,REG2
      if (type < 0)                       // если это не код операции,
                                          // является ли он директивой?
        type = search_pseudo_table(opcode);
      switch(type) {                      // определяем длину команды
        case 1:length=get_length_of_type1(line); break;
        case 2:length=get_length_of_type2(line); break;
        // другие варианты
    }
  }
  write_temp_file(type, opcode, length, line);  // информация для
                                                // второго прохода
  location_counter = location_counter + length; // обновление счетчика
                                                // адресов команд
  if (type == END_STATEMENT) {            // завершился ли ввод?
    more_input = false;                   // если да, то выполняем
                                          // служебные действия:
    rewind_temp_for_pass_two();           // перематываем файл обратно
    sort_literal_table();                 // сортируем таблицу литералов
    remove_redundant_literals();          // и удаляем из нее дубликаты
    }
  }
}
```

Некоторые процедуры будут относительно короткими — как, например, процедура `check_for_symbol`, которая просто возвращает в виде символьной строки имя, если таковое имеется, или ноль, если его нет. Другие процедуры, например `get_length_of_type1` и `get_length_of_type2`, могут быть достаточно длинными и сами вызывать процедуры. Естественно, на практике типов будет не два, а больше — это зависит от ассемблируемого языка и от того, сколько типов команд предусмотрено в этом языке.

Подобное структурирование программ имеет и другие преимущества, кроме простоты программирования. Если программа пишется группой людей, разнообразные процедуры могут быть поделены на фрагменты и распределены между программистами. Все подробности получения входных данных скрыты в процедуре `read_next_line`. Если эти детали нужно изменить (например, из-за изменений в операционной системе), то это повлияет только на одну подчиненную процедуру, и никаких изменений в самой процедуре `pass_one` вносить не нужно.

В процессе чтения программы во время первого прохода ассемблер должен проанализировать каждую строку, чтобы найти код операции (например, `ADD`), определить ее тип (набор операндов) и вычислить длину команды. Эта информация понадобится при втором проходе, поэтому ее лучше записать, чтобы не анализировать строку второй раз. Однако переписывание входного файла

потребует больше операций ввода-вывода. Что лучше — увеличить количество операций ввода-вывода, чтобы меньше времени тратить на анализ строк, или сократить количество операций ввода-вывода и потратить больше времени на анализ, зависит от быстродействия центрального процессора и дисковой памяти, эффективности файловой системы и некоторых других факторов. В нашем примере мы запишем временный файл с типом, кодом и длиной операции, а также саму входную строку. Именно эта строка и будет считываться при втором проходе, и читать файл по второму разу не потребуется.

После прочтения директивы `END` первый проход завершается. В этот момент можно сохранить таблицу символических имен и таблицу литералов, если это необходимо. В таблице литералов можно произвести сортировку и удалить дубликаты.

Второй проход

Цель второго прохода — генерирование объектной программы и вывод протокола ассемблирования (если нужно). Кроме того, при втором проходе должна выводиться информация, необходимая для компоновки в один исполняемый файл процедур, которые ассемблировались в разное время. В листинге 7.8 показана процедура для второго прохода.

Листинг 7.8. Второй проход простого ассемблера

```
public static void pass_two() {
  // Эта процедура – второй проход ассемблера
  boolean morejnput = true;           // флаг, останавливающий второй проход
  String line, opcode;                // поля команды
  int location_counter, length, type; // переменные
  final int END_STATEMENT = -2;       // сигналы окончания ввода
  final int MAX_CODE =16;             // максимальное число байтов в команде
  byte code[] = new byte[MAX_CODE];   // число байтов в команде
                                      // в сгенерированном коде

  location_counter = 0;               // ассемблирование первой
                                      // команды в адресе 0

  while (more_input) {                // more_input с помощью директивы END
                                      // получает значение "ложь"
    type = read_type();               // считывание поля типа следующей строки
    opcode = read_opcode();           // считывание поля кода операции
                                      // в следующей строке
    length = read_length();           // считывание поля длины
                                      // в следующей строке
    line = read_line();               // считывание самой входной строки

    if (type != 0) {                  // тип 0 указывает на комментарий
      switch(type) {                  // порождение выходного кода
        case 1:eval_type1(opcode, length, line, code); break;
        case 2: eval_type2(opcode, length, line, code); break;
        // другие варианты
      }
    }

    write_output(code);               // запись двоичного кода
```

продолжение ⇩

Листинг 7.8 *(продолжение)*

```
    write_listing(code, line);        // вывод на печать одной строки
    location_counter = location_counter + length; // обновление счетчика
                                                  // адресов команд
    if (type == END_STATEMENT) {      // завершен ли ввод?
      more_input = false;             // если да, то выполняем
                                      // служебные операции
      finish_up();                    // завершение
    }
  }
}
```

Процедура второго прохода похожа на процедуру первого: строки считываются по одной и обрабатываются тоже по одной. Поскольку мы записали в начале каждой строки тип, код операции и длину (во временном файле), все они считываются, и, таким образом, нам не нужно проводить анализ строк во второй раз. Основная работа по порождению кода выполняется процедурами `eval_type1`, `eval_type2` и т. д. Каждая из них обрабатывает определенную схему (например, код операции и два регистра-операнда). Полученный в результате двоичный код команды сохраняется в переменной `code`. Затем совершается контрольное считывание. Желательно, чтобы процедура `write_code` просто сохраняла в буфере накопленный двоичный код и записывала файл на диск большими кусками — это снизит нагрузку на диск.

Исходный оператор и выходной (объектный) код, полученный из него (в шестнадцатеричной системе), можно напечатать или поместить в буфер, чтобы напечатать потом. После смены значения счетчика адресов команд происходит выборка следующего оператора.

До сих пор мы считали, что исходная программа не содержит ошибок. Однако любой, кто хоть когда-нибудь занимался программированием, знает, насколько это предположение не соответствует действительности. Вот только наиболее распространенные ошибки:

- используемое символическое имя не определено;
- символическое имя определено более одного раза;
- имя в поле кода операции не является допустимым кодом операции;
- слишком мало операндов для данного кода операции;
- слишком много операндов для данного кода операции;
- восьмеричное число содержит цифру 8 или 9;
- недопустимое применение регистра (например, переход к регистру);
- отсутствует оператор `END`.

Программисты весьма изобретательны по части новых ошибок. Ошибки с неопределенным символическим именем часто являются следствием опечаток. Хороший ассемблер может просчитать, какой из всех определенных символов в большей степени соответствует неопределенному, и подставить его вместо неопределенного символа. Для исправления других ошибок ничего кардинального предложить нельзя. Лучшее, что может сделать ассемблер при обнаружении оператора с ошибкой, — вывести сообщение об ошибке на экран и попробовать продолжить ассемблирование.

Таблица символических имен

Во время первого прохода ассемблер накапливает всю информацию о символах и их значениях. Эту информацию он должен сохранить в таблице символических имен, к которой будет обращаться при втором проходе. Таблицу символических имен можно организовать несколькими способами. Некоторые из них мы опишем. При применении любого из этих способов мы пытаемся смоделировать **ассоциативную память**, которая представляет собой набор пар (символическое имя, значение). По имени ассоциативная память должна выдавать его значение.

Проще всего реализовать таблицу символических имен в виде массива пар, где первый элемент является именем (или указателем на имя), а второй — значением (или указателем на него). Если нужно найти какое-нибудь символическое имя, то таблица просто последовательно просматривается, пока не будет найдено соответствие. Такой метод довольно легко запрограммировать, но он медленно работает, поскольку при каждом поиске в среднем придется просматривать половину таблицы.

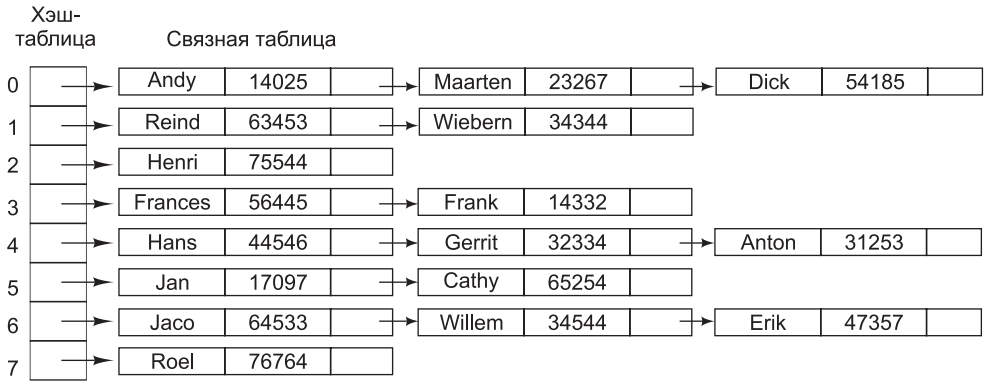
Другой способ организации — отсортировать таблицу по именам и для поиска имен использовать алгоритм **бинарного поиска**. В соответствии с этим алгоритмом средний элемент таблицы сравнивается с символическим именем. Если нужное имя по алфавиту идет раньше среднего элемента, значит, оно находится в первой половине таблицы. Если символическое имя по алфавиту идет после среднего элемента, значит, оно находится во второй части таблицы. Если нужное имя совпадает со средним элементом, то поиск на этом завершается.

Допустим, ключ среднего элемента таблицы не равен искомому символическому имени. Мы знаем, в какой половине таблицы он находится. Алгоритм двоичного поиска применяется к соответствующей половине. В результате мы либо получим совпадение, либо определим нужную четверть таблицы. Таким образом, в таблице из n элементов нужный символ можно найти примерно за $\log_2 n$ попыток. Очевидно, что такой алгоритм работает быстрее, чем просто последовательный просмотр таблицы, но при этом элементы таблицы нужно сохранять в алфавитном порядке.

Совершенно другой подход — **хеширование**. В этом случае используется хеш-функция, которая отображает символы (имена) на целые числа в промежутке от 0 до $k - 1$. Например, эта функция может перемножать ASCII-коды всех символов имени, игнорируя переполнение, а затем взять значение по модулю k или разделить полученное значение на простое число. Фактически подойдет любая входная функция, которая дает равномерное распределение значений. Символические имена можно хранить в таблице, состоящей из k сегментов, от 0 до $k - 1$. Все пары (символическое имя, значение), в которых имя соответствует i, сохраняются в связном списке, на который указывает слот i в хеш-таблице. Если в хеш-таблице содержится n символических имен и k слотов, то в среднем длина списка будет n/k. Если мы выберем k, приблизительно равное n, то на нахождение нужного символического имени в среднем потребуется всего один поиск. Путем корректировки k мы можем сократить размер таблицы, но при этом скорость поиска снизится. Процесс хеширования иллюстрирует рис. 7.1.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebern	34344	1

а



б

Рис. 7.1. Хеширование: символические имена, значения и хеш-коды, образованные от символических имен (*а*); хеш-таблица из 8 элементов со связным списком символических имен и значений (*б*)

Компоновка и загрузка

Большинство программ содержат более одной процедуры. Компиляторы и ассемблеры транслируют одну процедуру и записывают полученный результат на диск. Перед запуском программы должны быть найдены и скомпонованы все оттранслированные процедуры. Если виртуальная память недоступна, скомпонованная программа должна загружаться непосредственно в основную память. Программы, которые выполняют эти функции, называются по-разному: **компоновщиками**, **компонующими загрузчиками**, **редакторами связей**. Для полной трансляции исходной программы требуется два шага (рис. 7.2):

1. Компиляция или ассемблирование исходных процедур.
2. Компоновка объектных модулей.

Первый шаг выполняется ассемблером или компилятором, второй — компоновщиком.

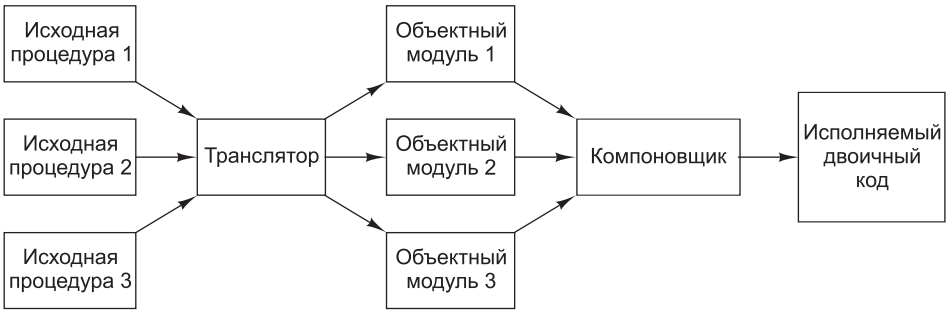


Рис. 7.2. Для получения исполняемой двоичной программы из совокупности оттранслированных независимо друг от друга процедур используется компоновщик

Трансляция исходной процедуры в объектный модуль — это переход на другой уровень, поскольку исходный и выходной языки имеют разные команды и синтаксис. Однако при компоновке перехода на другой уровень не происходит, поскольку программы на входе и выходе компоновщика предназначены для одной и той же виртуальной машины. Цель компоновщика — скомпоновать вместе все процедуры, которые транслировались отдельно, чтобы в результате получился **исполняемый двоичный код**. В системах семейства Windows объектные модули имеют расширение .obj, а исполняемые программы — .exe. В UNIX объектные модули имеют расширение .o, а исполняемые программы расширения не имеют.

Компиляторы и ассемблеры транслируют каждую исходную процедуру как отдельную единицу. На это есть веская причина. Если компилятор или ассемблер считывал бы целый ряд исходных процедур и сразу переводил бы их в готовую программу на машинном языке, то при изменении одного оператора в исходной процедуре потребовалось бы заново транслировать все исходные процедуры.

Если каждая процедура транслируется отдельно, как показано на рис. 7.2, то транслировать заново придется только одну измененную процедуру, хотя понадобится заново скомпоновать все объектные модули. Однако компоновка происходит гораздо быстрее, чем трансляция, поэтому при доработке программы оба шага (трансляция и компоновка) выполняются быстро. Это особенно важно для программ, содержащих сотни или тысячи модулей.

Задачи компоновщика

В начале первого прохода ассемблирования счетчик адресов команд устанавливается на 0. Этот шаг эквивалентен предположению, что объектный модуль во время выполнения будет находиться в ячейке с адресом 0. На рис. 7.3 показаны 4 объектных модуля, подготовленных для типичной машины. В этом примере каждый модуль начинается с команды перехода `BRANCH` к команде `MOVE` в том же модуле.

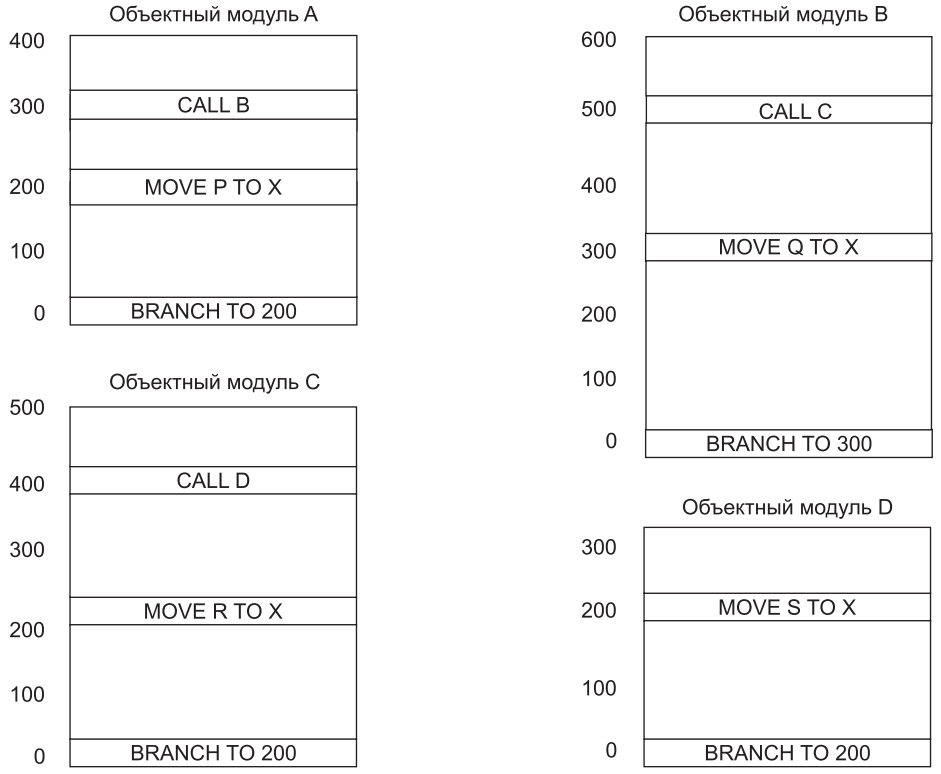


Рис. 7.3. Каждый модуль имеет собственное адресное пространство, начинающееся с нулевого адреса

Чтобы запустить программу, компоновщик помещает объектные модули в основную память, формируя образ исполняемого двоичного кода (рис. 7.4, *a*). Цель — создать точный образ виртуального адресного пространства исполняемой программы внутри компоновщика и разместить все объектные модули по соответствующим адресам. Если физической или виртуальной памяти для формирования образа недостаточно, можно использовать дисковый файл. Обычно небольшой раздел памяти, начинающийся с нулевого адреса, используется для векторов прерываний, взаимодействия с операционной системой, обнаружения неинициализированных указателей и других целей, поэтому, как правило, программы начинаются не с нулевого адреса, а выше. В нашем примере программы начинаются с (произвольно выбранного) адреса 100.

Посмотрите на рис. 7.4, *a*. Хотя программа уже загружена в образ исполняемого двоичного файла, она еще не готова для выполнения. Посмотрим, что произойдет, если выполнение программы начнется с команды в начале модуля *A*. Программа не совершит перехода к команде `MOVE`, поскольку эта команда находится в ячейке с адресом 300. Фактически все команды обращения к памяти по той же причине выполнены не будут.

Здесь возникает **проблема перераспределения памяти**, поскольку каждый объектный модуль на рис. 7.3 занимает собственное адресное пространство. В машине с сегментированным адресным пространством (например, на платформе

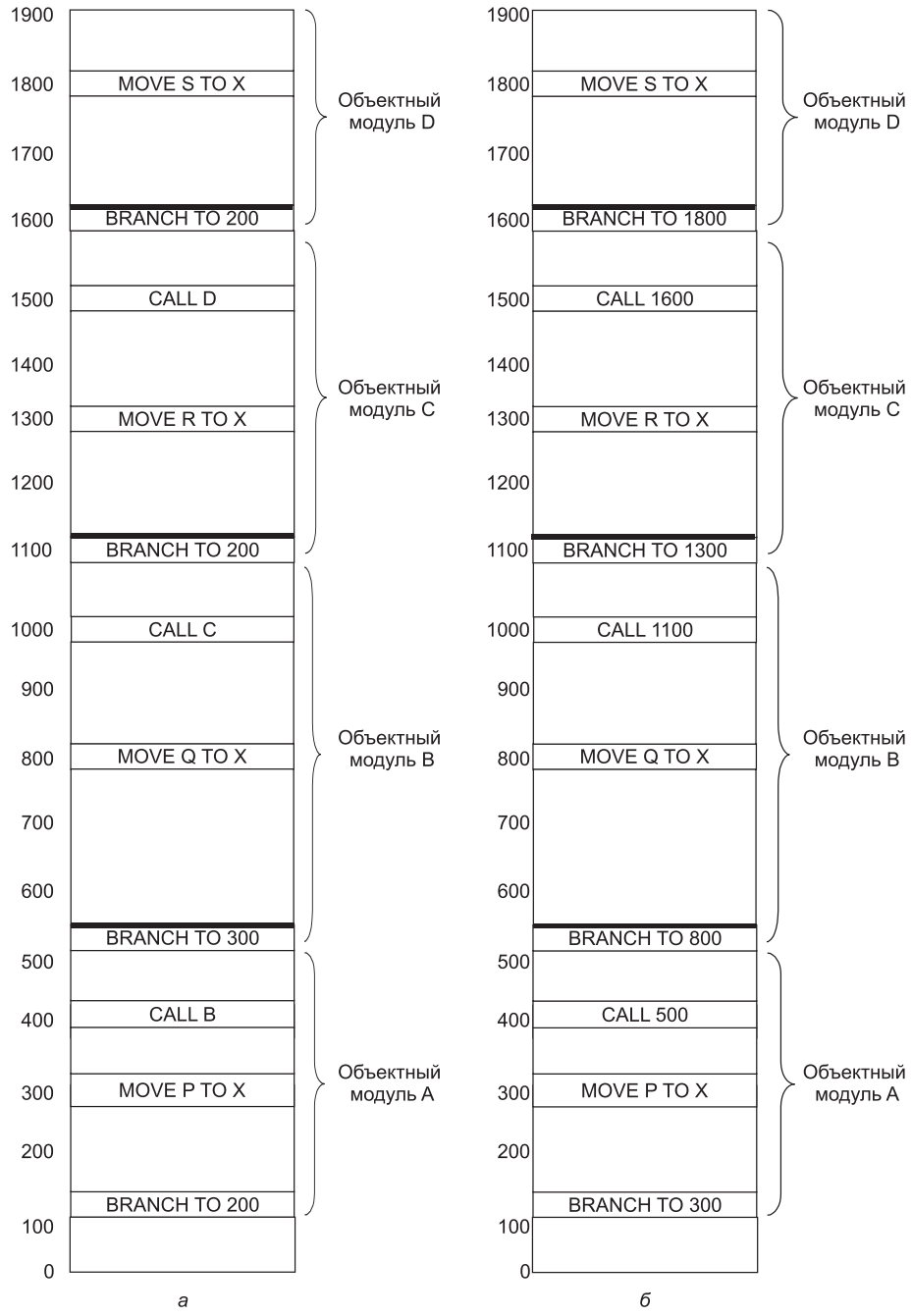


Рис. 7.4. Объектные модули после размещения в двоичном образе, но до перераспределения памяти и компоновки (*а*); те же объектные модули после компоновки и перераспределения памяти (*б*). В результате получается исполняемый двоичный код, который можно запускать

x86) каждый объектный модуль теоретически может иметь собственное адресное пространство, если его поместить в отдельный сегмент. Однако для x86 только система OS/2 поддерживала такую возможность. Все версии Windows и UNIX поддерживают одно линейное адресное пространство, поэтому все объектные модули должны быть объединены в одном адресном пространстве.

Более того, команды вызова процедур на рис. 7.4, *а* вообще не будут работать. В ячейке с адресом 400 программист намеревается вызвать объектный модуль *B*, но поскольку каждая процедура транслируется отдельно, ассемблер не может определить, какой адрес вставлять в команду `CALL B`, поскольку адрес объектного модуля *B* до компоновки неизвестен. Эта проблема называется **проблемой внешней ссылки**. Обе проблемы решаются с помощью компоновщика.

Компоновщик объединяет отдельные адресные пространства объектных модулей в единое линейное адресное пространство. Для этого совершаются следующие шаги:

1. Компоновщик строит таблицу объектных модулей и их размеров.
2. На основе этой таблицы он приписывает начальные адреса каждому объектному модулю.
3. Компоновщик находит все команды, которые обращаются к памяти, и прибавляет к каждой из них **константу перераспределения**, равную начальному адресу этого модуля.
4. Компоновщик находит все команды, которые обращаются к процедурам, и вставляет в них адреса этих процедур.

Таблица 7.5 соответствует построенной на первом шаге таблице объектных модулей, представленных рис. 7.4. В ней даются имя, длина и начальный адрес каждого модуля.

Таблица 7.5. Имя, длина и начальный адрес каждого модуля на рис. 7.4

Модуль	Длина	Начальный адрес
A	400	100
B	600	500
C	500	1100
D	300	1600

На рис. 7.4, *б* показано, как выглядит адресное пространство после завершения работы компоновщика.

Структура объектного модуля

Объектные модули обычно состоят из шести частей:

1. Идентификация.
2. Таблица точек входа.
3. Таблица внешних ссылок.
4. Машинные команды и константы.
5. Словарь перераспределения.
6. Конец модуля.

В первой части содержатся имя модуля, некоторая информация, необходимая компоновщику (например, данные о длине различных частей модуля), а иногда дата ассемблирования.

Вторая часть объектного модуля — это список символов, определенных в модуле, вместе с их значениями. К этим символам могут обращаться другие модули. Например, если модуль состоит из процедуры `bigbug`, то элемент таблицы будет содержать символьную строку `"bigbug"` с соответствующим адресом. Программист, пишущий на языке ассемблера, с помощью директивы `PUBLIC` указывает, какие символические имена считаются **точками входа**.

Третья часть объектного модуля состоит из списка символических имен, которые применяются в данном модуле, а определены в других модулях. Здесь также имеется еще один список, который показывает, какие именно символические имена используются теми или иными машинными командами. Второй список нужен для того, чтобы компоновщик мог вставлять правильные адреса в команды, которые используют внешние имена. Процедура может вызывать другие независимо транслируемые процедуры, объявив имена вызываемых процедур внешними. Программист, пишущий на языке ассемблера, с помощью директивы `EXTERN` указывает, какие символы нужно объявить **внешними**. В некоторых компьютерах точки входа и внешние ссылки объединены в одной таблице.

Четвертая часть объектного модуля — машинные команды и константы. Это единственная часть объектного модуля, которая загружается в память для выполнения. Остальные пять частей используются компоновщиком, а затем отбрасываются еще до начала выполнения программы.

Пятая часть объектного модуля — это словарь перераспределения памяти. К командам, которые содержат адреса памяти, должна прибавляться константа перераспределения (см. рис. 7.4). Компоновщик сам не может определить, какие слова в части 4 содержат машинные команды, а какие — константы. Поэтому в этой таблице содержится информация о том, какие адреса нужно перераспределять. Это может быть битовая таблица, где на каждый бит приходится потенциально перераспределяемый адрес, либо явный список перераспределяемых адресов.

Шестая часть содержит указание на конец модуля, а иногда — контрольную сумму для определения ошибок, сделанных во время чтения модуля, и адрес, с которого должно начинаться исполнение.

Большинству компоновщиков требуются два прохода. На первом проходе компоновщик считывает все объектные модули и строит таблицу имен и размеров модулей, а также глобальную таблицу символов, которая состоит из всех точек входа и внешних ссылок. На втором проходе модули поочередно считываются, перераспределяются в памяти и компонуются.

Время компоновки и динамическое перераспределение памяти

В мультипрограммной системе программу можно считать в основную память, запустить на некоторое время, записать на диск, а затем снова считать в основную память для выполнения. В сложной системе с большим количеством программ трудно быть уверенным, что программа считывается каждый раз в одно и то же место в памяти.

На рис. 7.5 показано, что произойдет, если уже распределенная в памяти программа (см. рис. 7.4, *б*) будет загружена по адресу 400, а не по адресу 100, куда ее изначально поместил компоновщик. Все адреса памяти оказываются неправильными, и это при отсутствии какой-либо информации о распределении памяти.

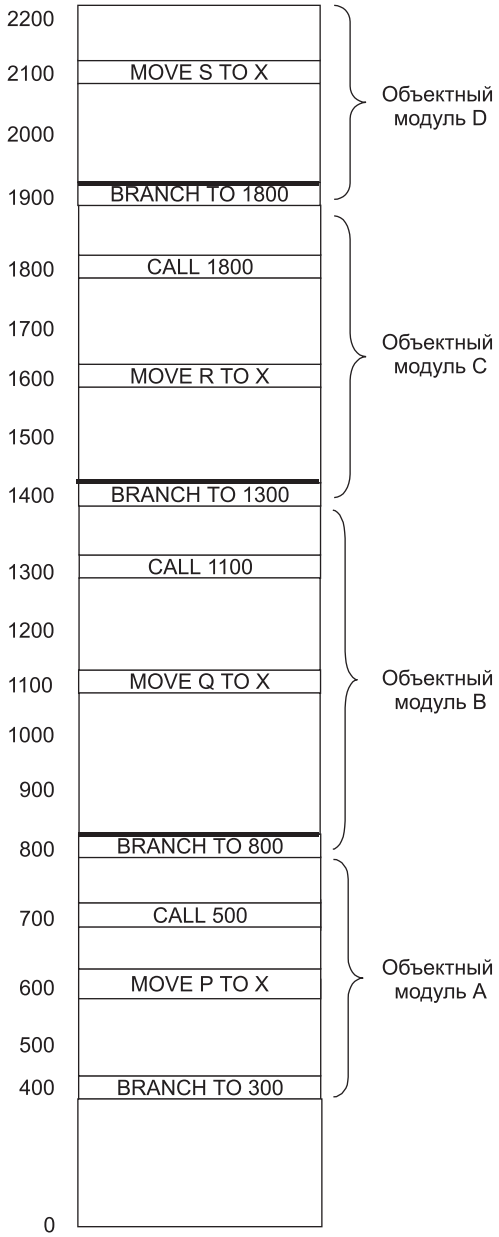


Рис. 7.5. Исполняемая программа с рис. 7.4, *б*, сдвинутая вверх на 300 адресов. Многие команды теперь обращаются по неправильным адресам памяти

Даже если эта информация была бы доступна, перераспределять все адреса при каждой загрузке программы было бы расточительно.

Проблема перемещения программ, уже скомпонованных и размещенных в памяти, имеет непосредственное отношение к моменту окончательного связывания символических имен с абсолютными адресами физической памяти. В программе содержатся символические имена для адресов памяти (например, `BR L`). Время, в которое определяется адрес в основной памяти, соответствующий имени `L`, называется **временем связывания**. Существуют, по крайней мере, шесть вариантов времени связывания:

1. Когда программа пишется.
2. Когда программа транслируется.
3. Когда программа компонуется, но еще до загрузки.
4. Когда программа загружается.
5. Когда загружается базовый регистр, который используется для адресации.
6. Когда выполняется команда, содержащая требуемый адрес.

Если команда, содержащая адрес, после связывания перераспределяется в памяти, этот адрес оказывается неправильным (предполагается, что объект, на который происходит ссылка, тоже перемещается в памяти). Если транслятор генерирует исполняемый двоичный код, то связывание происходит во время трансляции, и программа должна быть запущена с адреса, указанного транслятором. При применении метода, описанного в предыдущем подразделе, символические имена в процессе компоновки связываются с абсолютными адресами, и именно по этой причине перемещать программы после компоновки нельзя (см. рис. 7.5).

Здесь возникают два вопроса. Во-первых, когда символические имена связываются с виртуальными адресами? Во-вторых, когда виртуальные адреса связываются с физическими адресами? Только после двух этих операций процесс связывания можно считать завершенным. Когда компоновщик объединяет отдельные адресные пространства объектных модулей в единое линейное адресное пространство, он фактически создает виртуальное адресное пространство. Перераспределение в памяти и компоновка нужны для связывания символических имен с определенными виртуальными адресами. Это верно независимо от того, используется виртуальная память или нет.

Предположим, что адресное пространство, изображенное на рис. 7.4, *б*, было разбито на страницы. Ясно, что виртуальные адреса, соответствующие символическим именам *A*, *B*, *C* и *D*, уже определены, хотя их физические адреса будут зависеть от содержания таблицы страниц. Реально связывание символических имен с виртуальными адресами происходит в исполняемом двоичном коде.

Любой механизм, позволяющий легко изменять отображение виртуальных адресов на адреса основной физической памяти, упростит перемещение программы в основной памяти, даже если они уже связаны с виртуальным адресным пространством. Одним из таких механизмов является разбиение на страницы. Если программа перемещается в основной памяти, потребуется изменить только ее таблицу страниц, а не саму программу.

Второй механизм — применение регистра перераспределения времени исполнения. Компьютер CDC 6600 и его последователи содержали такой регистр.

В машинах, в которых используется эта технология перераспределения памяти, регистр всегда указывает на физический адрес начала текущей программы. Аппаратно этот регистр добавляется ко всем адресам перед их распределением в памяти. Весь процесс перераспределения памяти прозрачен для пользовательских программ. Пользовательские программы даже не подозревают, что происходит перераспределение памяти. Если программа перемещается, операционная система должна обновить регистр перераспределения. Такой механизм менее распространен, чем разбиение на страницы, поскольку перемещаться должна вся программа целиком (если есть отдельные регистры перераспределения для кода и данных, как, например, в процессоре Intel 8088, то объектами перемещения будут обе части программы).

Третий механизм можно использовать в машинах, в которых реализована возможность обращения к памяти относительно счетчика команд. В этих машинах всякий раз, когда программа перемещается в основной памяти, достаточно обновить только счетчик команд. Программа, все обращения к памяти которой либо связаны со счетчиком команд, либо абсолютны (например, обращения по абсолютным адресам к регистрам устройств ввода-вывода), называется **позиционно-независимой**. Позиционно-независимую процедуру можно поместить в любом месте виртуального адресного пространства без необходимости перераспределять адреса памяти.

Динамическая компоновка

Стратегия компоновки, которую мы обсуждали в подразделе «Задачи компоновщика», имеет одну особенность: все процедуры, требуемые программе, компонуются до начала работы программы. Однако если все связи будут устанавливаться до начала работы программы в компьютере с виртуальной памятью, возможности виртуальной памяти будут использоваться неполностью. Многие программы содержат процедуры, которые вызываются только при определенных обстоятельствах. Например, компиляторы содержат процедуры для компиляции редко используемых операторов или исправления редко встречающихся ошибок.

Более гибкий способ компоновки раздельно скомпилированных процедур — компоновка каждой процедуры в момент ее первого вызова. Этот процесс называется **динамической компоновкой**. Впервые динамическая компоновка была применена в системе MULTICS, причем в некоторых отношениях эта реализация так и остается непревзойденной. Давайте рассмотрим примеры динамической компоновки в нескольких системах.

Динамическая компоновка в MULTICS

В системе MULTICS с каждой программой соотносится сегмент, так называемый **сегмент компоновки** (linkage segment). Он содержит один информационный блок для каждой процедуры, которая может быть вызвана. Этот блок начинается со слова, зарезервированного для виртуального адреса процедуры, за ним следует имя процедуры, которое сохраняется в виде символьной строки.

При динамической компоновке вызовы процедур во входном языке транслируются в команды, которые путем косвенной адресации обращаются к первому слову соответствующего блока, как показано на рис. 7.6, *а*. Компилятор заполняет это слово либо недействительным адресом, либо специальным набором битов, который вызывает исключение.

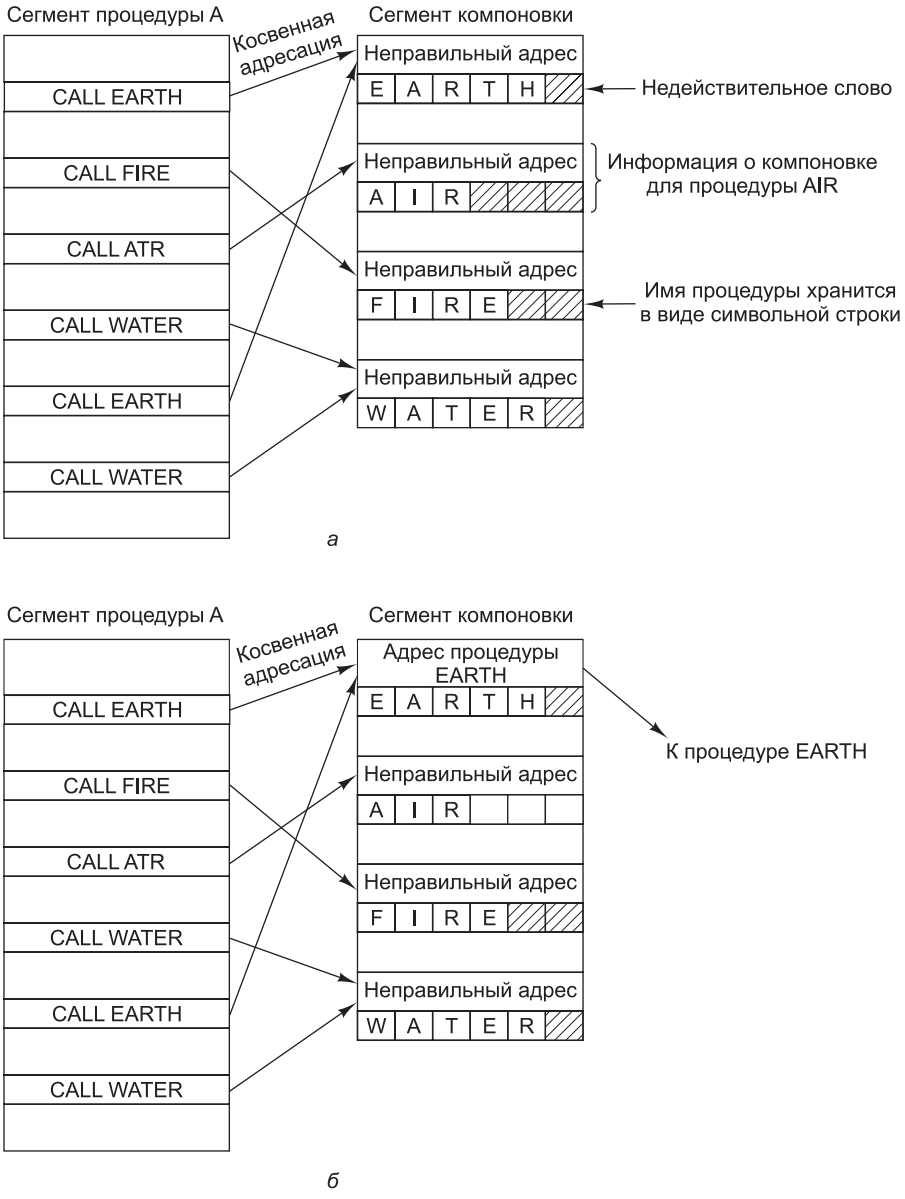


Рис. 7.6. Динамическая компоновка: процедура EARTH до вызова (*а*); процедура EARTH после вызова и компоновки (*б*)

При вызове процедуры в другом сегменте попытка косвенного обращения к недействительному слову вызывает исключение компоновщика. Затем компоновщик находит символьную строку в слове, которое следует за неправильным адресом, и начинает искать пользовательский каталог для скомпилированной процедуры с найденным именем. Далее этой процедуре выделяется виртуальный адрес (обычно в ее собственном сегменте), и этот виртуальный адрес записывается

поверх неправильного адреса, как показано на рис. 7.6, *б*. После этого команда, которая вызвала ошибку компоновки, выполняется заново, что позволяет программе продолжать работу с того места, где она находилась до исключения.

Все последующие обращения к этой процедуре будут выполняться без ошибок, поскольку недействительное ранее слово теперь содержит правильный виртуальный адрес. Следовательно, при динамической компоновке компоновщик вызывается только тогда, когда процедура вызывается впервые. Вызывать его повторно уже не нужно.

Динамическая компоновка в Windows

Все версии Windows, в том числе Windows NT, поддерживают динамическую компоновку. При динамической компоновке используется специальный файловый формат, который называется **DLL** (Dynamic Link Library — **библиотека динамической компоновки**). Библиотеки динамической компоновки могут содержать процедуры, данные или то и другое вместе. Обычно они применяются для того, чтобы два и более процессов могли разделять процедуры и данные библиотеки. Большинство DLL-файлов имеют расширение .dll, но встречаются и другие расширения, например .drv (для библиотек драйверов — driver libraries) и .fon (для библиотек шрифтов — font libraries).

Самая распространенная форма DLL — библиотека, состоящая из набора процедур, которые могут загружаться в память и к которым имеют доступ несколько процессов одновременно. На рис. 7.7 показаны два процесса, которые совместно используют DLL-файл, содержащий 4 процедуры, *A*, *B*, *C* и *D*. Программа 1 использует процедуру *A*; программа 2 — процедуру *C*, хотя они вполне могли бы использовать одну и ту же процедуру.

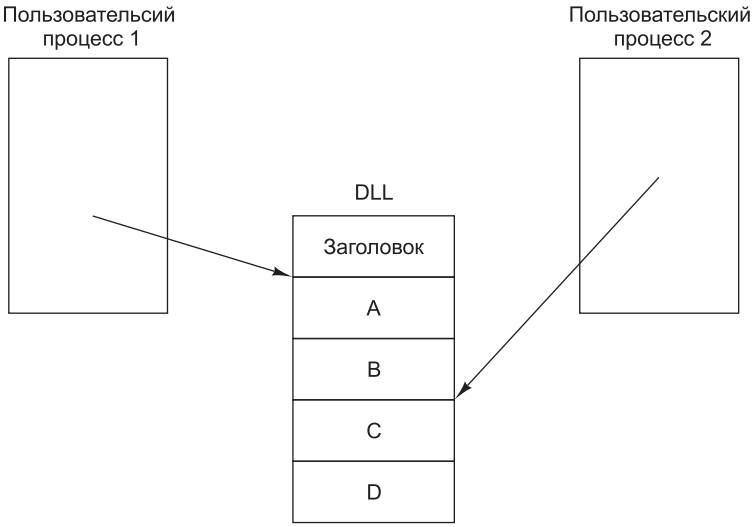


Рис. 7.7. Два процесса используют один DLL-файл

DLL-файл строится компоновщиком из набора входных файлов. Построение DDL напоминает создание исполняемого двоичного кода, только при создании DLL компоновщику передается специальный флаг, сообщающий о том, что

требуется именно библиотека DLL. DLL-файлы обычно собираются из набора библиотечных процедур, которые могут понадобиться нескольким процессам. Типичными примерами DLL-файлов являются процедуры сопряжения с библиотекой системных вызовов Windows и большими графическими библиотеками. Применяя DDL-файлы, мы экономим пространство в памяти и на диске. Если бы та или иная библиотека была связана статически с каждой использующей ее программой, эту библиотеку пришлось бы включать во все исполняемые двоичные программы в памяти и на диске, что было бы слишком расточительно. А при наличии DLL-файла на диске и в памяти будет находиться всего один экземпляр библиотеки.

Кроме того, такой подход упрощает обновление библиотечных процедур, причем даже после того, как использующие их программы скомпилированы и скомпонованы. Для коммерческих программных пакетов, где входная программа пользователям обычно недоступна, наличие DLL-файлов означает, что поставщик программного обеспечения сможет исправлять обнаруженные программные ошибки, просто распространяя новые DLL-файлы через Интернет, причем при этом не потребуется никаких изменений в двоичных кодах основных программ.

Основное отличие между DLL и исполняемой двоичной программой состоит в том, что DLL-файл не может запускаться и работать сам по себе (поскольку у него нет главной программы). Он также содержит совершенно другую информацию в заголовке. Кроме того, DLL-файл имеет несколько дополнительных процедур, не связанных с процедурами в библиотеке. Например, существует одна процедура, которая вызывается автоматически всякий раз, когда новый процесс связывается с DLL-файлом, и еще одна процедура, которая вызывается автоматически всякий раз, когда связь процесса с DLL-файлом разрывается. Эти процедуры могут выделять и освобождать память или управлять другими ресурсами, необходимыми DLL-файлу.

Программа может установить связь с DLL-файлом двумя способами: путем неявной или явной компоновки. При **неявной компоновке** пользовательская программа статически компонуется со специальным файлом, так называемой **библиотекой импорта**. Библиотека импорта создается специальной утилитой, предназначенной для извлечения определенной информации из DLL-файла. Библиотека импорта предоставляет связующее звено, через которое пользовательская программа получает доступ к DLL-файлу. Пользовательская программа может быть связана с несколькими библиотеками импорта. Когда программа, в которой имеет место неявная компоновка, загружается в память для исполнения, Windows проверяет, какие DLL-файлы ей требуются и все ли они находятся в памяти. Те файлы, которых еще нет в памяти, немедленно туда загружаются (но необязательно целиком, поскольку они разбиты на страницы). Затем производятся определенные изменения структур данных в библиотеках импорта так, чтобы можно было определить местоположение вызываемых процедур (это похоже на изменения, которые иллюстрирует рис. 7.6). Их тоже нужно отобразить на виртуальное адресное пространство программы. С этого момента пользовательская программа готова к запуску; она может вызывать процедуры из DLL-файлов так, словно они статически с ней связаны.

Альтернативой является **явная компоновка**. Явная компоновка не требует ни библиотек импорта, ни одновременной загрузки DLL-файлов с пользователь-

ской программой. Вместо этого пользовательская программа совершает явный вызов прямо во время работы, чтобы установить связь с DLL-файлом, а затем совершает дополнительные вызовы для получения адресов процедур, которые ей требуются. Когда все это сделано, программа совершает финальный вызов, чтобы разорвать связь с DLL-файлом. Когда последний процесс разрывает связь с DLL-файлом, этот файл может быть выгружен из памяти.

Важно понимать, что процедура в DLL-файле не имеет отличительных признаков (как процессы или программные потоки). Она работает в потоке вызывающей программы и для своих локальных переменных использует стек вызывающей программы. Она может содержать статические данные, специфичные для процесса (а также общие), а в остальном работает, как статически скомпонованная процедура. Единственным существенным отличием является способ установления связи.

Динамическая компоновка в UNIX

В UNIX используются **библиотеки коллективного доступа**, по сути напоминающие библиотеки DLL Windows. Как и DLL-файл, библиотека коллективного доступа представляет собой архивный файл, содержащий несколько процедур или модулей данных, которые присутствуют в памяти во время работы программы и одновременно могут быть связаны с несколькими процессами. Стандартная библиотека C и большинство сетевых программ являются библиотеками коллективного доступа.

UNIX поддерживает только неявную компоновку, поэтому библиотека коллективного доступа состоит из двух частей: **главной библиотеки** (host library), которая статически скомпонована с исполняемым файлом, и **целевой библиотеки** (target library), которая вызывается во время работы программы. Несмотря на некоторые различия в деталях, по существу эта концепция соответствует концепции DLL.

Краткое содержание главы

Хотя большинство программ можно и нужно писать на языках высокого уровня, в некоторых ситуациях приходится применять язык ассемблера. Вероятные кандидаты — программы для компьютеров с недостаточным объемом ресурсов (например, процессоры для смарт-карт, различных приборов, портативных цифровых устройств). Ассемблерная программа — это символьное представление программы на том или ином машинном языке. Она транслируется на машинный язык специальной программой, которая называется ассемблером.

Если для какого-либо приложения требуется высокое быстродействие, лучше всего сначала написать программу на языке высокого уровня, затем путем тестов установить, исполнение какой части программы занимает большую часть времени, и переписать на ассемблере только эти части программы. Практика показывает, что часто небольшая часть всей программы занимает большую часть всего времени выполнения этой программы.

Во многих ассемблерах предусмотрены макросы, которые позволяют программистам давать символические имена целым последовательностям команд.

Обычно эти макросы могут быть параметризированы. Макросы реализуются с помощью алгоритма обработки строковых литералов.

Большинство ассемблеров двухпроходные. Во время первого прохода строится таблица символических имен для меток, литералов и объявляемых идентификаторов. Символические имена можно либо не сортировать и искать путем последовательного просмотра таблицы, либо сначала сортировать, а потом применять двоичный поиск, либо хешировать. Если символические имена не нужно удалять во время первого прохода, хеширование — это лучший метод. Исполняемая программа создается во время второго прохода. Что касается директив (псевдокоманд), то одни из них выполняются при первом проходе, другие — при втором.

Программы, которые ассемблируются независимо друг от друга, можно скомпоновать вместе и получить исполняемую двоичную программу. Эту работу выполняет компоновщик, обеспечивающий перераспределение программ в памяти и связывание имен. Динамическая компоновка — это технология, при которой определенные процедуры не компонуются до момента вызова. Библиотеки коллективного доступа в UNIX и библиотеки динамической компоновки (DLL) в Windows используют технологию динамической компоновки.

Вопросы и задания

1. В некоторой программе всего 2 % кода занимают 50 % времени выполнения. Сравните следующие три стратегии с точки зрения времени программирования и выполнения. Предположим, что для написания программы на языке C требуется 100 человеко-месяцев, а программу на ассемблере писать в 10 раз труднее, но зато она работает в 4 раза эффективнее.
   1) Вся программа пишется на C.
   2) Вся программа пишется на ассемблере.
   3) Программа сначала пишется на C, а затем нужные 2 % программы переписываются на ассемблере.
2. Применимы ли к компиляторам правила работы двухпроходных ассемблеров? Рассмотрите следующие гипотетические ситуации.
   1) Компилятор вместо ассемблерного кода генерирует объектные модули.
   2) Компилятор генерирует символический язык ассемблера.
3. Все ассемблеры для платформы x86 в качестве первого операнда получают целевой адрес, а в качестве второго — исходный адрес. Какие проблемы могут возникнуть при другом подходе?
4. Можно ли следующую программу ассемблировать в два прохода? `EQU` — это директива, которая приравнивает метку к выражению в поле операнда.
```
P EQU Q
Q EQU R
R EQU S
S EQU 4
```
5. Некая компания планирует разработать ассемблер для компьютера с 40-разрядным словом. Чтобы снизить стоимость, менеджер проекта решил огра-

ничить длину символических имен, чтобы каждое имя можно было хранить в одном слове. Менеджер объявил, что символические имена могут состоять только из букв, причем буква *Q* запрещена. Какова максимальная длина символического имени? Опишите вашу схему кодирования.

6. Чем отличается команда от директивы?
7. Чем отличается счетчик адресов команд от счетчика команд? А существует ли вообще между ними различие? Ведь тот и другой отслеживают информацию о следующей команде в программе.
8. Какой будет таблица символических имен после обработки следующих операторов ассемблера x86 (первому оператору приписан адрес 1000)?

```
EVEREST:  POP BX         ; (1 байт)
K2:       PUSH BP        ; (1 байт)
WHITNEY:  MOV BP,SP      ; (2 байта)
MCKINLEY: PUSH X         ; (3 байта)
FUJI:     PUSH SI        ; (1 байт)
KIBO:     SUB SI,300     ; (3 байта)
```

9. Можете ли вы представить себе обстоятельства, при которых метка совпадет с кодом операции (например, могут ли команды `MOV` использоваться в качестве метки)? Аргументируйте.
10. Какие шаги нужно совершить, чтобы путем двоичного поиска найти элемент «Berkeley» в следующем списке: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood, Yellow Springs. Когда будете вычислять средний элемент в списке из четного числа элементов, возьмите элемент, который идет сразу после среднего.
11. Можно ли использовать двоичный поиск в таблице, в которой содержится простое число элементов?
12. Вычислите хеш-код для каждого из следующих символических имен. Для этого сложите буквы (a = 1, b = 2 и т. д.) и возьмите результат по модулю размера хеш-таблицы. Хеш-таблица содержит 19 слотов (от 0 до 18).

    els, jan, jelle, maaike

    Обеспечивает ли каждое символическое имя уникальное значение хеш-функции? Если нет, то как разрешить коллизию?
13. Метод хеширования, описанный в тексте главы, позволяет скомпоновать все элементы, имеющие один хеш-код, в связном списке. Альтернативный метод — создание только одной таблицы из n слотов, в которой в каждом слоте имеется пространство для одного ключа и его значения (или для указателей на них). Если алгоритм хеширования порождает слот, который уже заполнен, производится вторая попытка с использованием того же алгоритма хеширования. Если и на этот раз слот заполнен, алгоритм применяется снова и т. д. Так продолжается до тех пор, пока не будет найден пустой слот. Если доля слотов, которые уже заполнены, составляет R, то сколько попыток в среднем понадобится для того, чтобы ввести в таблицу новый символ?
14. Вероятно, когда-нибудь в будущем на одну микросхему можно будет помещать тысячи идентичных процессоров, каждый из которых содержит несколько слов локальной памяти. Если все процессоры могут считывать и записывать три общих регистра, то как реализовать ассоциативную память?

15. Процессор x86 имеет сегментированную архитектуру с несколькими независимыми сегментами. Ассемблер для этой машины может содержать директиву `SEG N`, которая помещает последующий код и данные в сегмент `N`. Повлияет ли такая схема на счетчик адресов команд?
16. Программы часто компонуются с многочисленными библиотеками DLL. А не будет ли более эффективным просто поместить все процедуры в один большой DLL-файл, а затем скомпоновать его?
17. Можно ли отобразить DLL-файл на виртуальные адресные пространства двух процессов с разными виртуальными адресами? Если да, то какие проблемы при этом возникают? Можно ли их разрешить? Если нет, то что можно сделать, чтобы устранить их?
18. Один из способов (статической) компоновки выглядит следующим образом: перед сканированием библиотеки компоновщик составляет список необходимых процедур, то есть имен, которые в компонуемых модулях определены как внешние (`EXTERN`). Затем компоновщик последовательно просматривает всю библиотеку, извлекая каждую процедуру, которая находится в списке нужных имен. Будет ли работать такая схема? Если нет, то почему и как это можно исправить?
19. Может ли регистр использоваться в качестве фактического параметра в макровызове? А константа? Если да, то почему? Если нет, то почему?
20. Вам нужно реализовать макроассемблер. Из эстетических соображений ваш начальник решил, что макроопределения не должны предшествовать вызовам макросов. Как повлияет это решение на реализацию?
21. Подумайте, как можно ввести макроассемблер в бесконечный цикл.
22. Компоновщик считывает 5 модулей размером 200, 800, 600, 500 и 700 слов соответственно. Если они загружаются в этом порядке, то каковы константы перераспределения памяти?
23. Напишите модуль таблицы символических имен, состоящий из двух процедур: `enter(symbol, value)` и `lookup(symbol, value)`. Первая вводит новые символические имена в таблицу, а вторая ищет их в таблице. Используйте какой-либо вариант хеширования.
24. Повторите предыдущее упражнение, но вместо хеш-таблицы после ввода последнего имени отсортируйте таблицу и используйте для поиска символических имен алгоритм бинарного поиска.
25. Напишите простой ассемблер для компьютера Mic-1, о котором мы говорили в главе 4. Помимо оперирования машинными командами, обеспечьте возможность приписывать константы символическим именам во время ассемблирования, а также возможность ассемблировать константу в машинное слово.
26. Добавьте макросы к ассемблеру, который вы должны были написать, выполняя предыдущее задание.

Глава 8
Параллельные компьютерные архитектуры

Скорость работы компьютеров становится все выше, но и предъявляемые к ним требования постоянно растут. Астрономы пытаются смоделировать всю историю Вселенной с момента большого взрыва и до сегодняшнего дня. Фармацевты хотели бы разрабатывать новые лекарственные препараты с помощью компьютеров, не принося в жертву легионы лабораторных крыс. Разработчики летательных аппаратов могли бы получать лучшие результаты, если бы вместо строительства огромных аэродинамических труб моделировали свои конструкции на компьютере. Если говорить коротко, какими бы мощными ни были компьютеры, их возможностей никогда не хватит для решения многих нетривиальных задач (особенно научных, технических и промышленных).

Хотя тактовая частота постоянно растет, производительность элементной базы нельзя увеличивать бесконечно. Главной проблемой остается скорость света — невозможно заставить протоны и электроны двигаться быстрее. Из-за высокой теплоотдачи компьютеры превратились в сверхтехнологичные кондиционеры. Наконец, поскольку размеры транзисторов постоянно уменьшаются, в конце концов наступит время, когда каждый транзистор будет состоять из нескольких атомов, поэтому основной проблемой могут стать законы квантовой механики (например, принцип неопределенности Гейзенберга).

В результате, чтобы иметь возможность решать более сложные задачи, разработчики обратились к компьютерам параллельного действия (далее — параллельные компьютеры). Невозможно построить компьютер с одним процессором и временем цикла в 0,001 нс, но зато можно построить компьютер с 1000 процессорами, время цикла каждого из которых составляет 1 нс. И хотя быстродействия каждого процессора во втором случае очевидно мало, теоретически мы должны получить требуемую производительность.

Параллелизм можно вводить на разных уровнях. На самом низком уровне он может быть реализован в процессоре за счет конвейеризации и суперскалярной архитектуры с несколькими функциональными блоками. Скрытого параллелизма можно добиться путем значительного удлинения слов в командах. Посредством дополнительных функций можно «научить» процессор одновременно обрабатывать несколько программных потоков. Наконец, можно установить на одной микросхеме несколько процессоров. Однако все эти приемы, вместе взятые, способны повысить производительность максимум в 10 раз по сравнению с классическими последовательными решениями.

На следующем уровне возможно внедрение в систему внешних плат ЦП с улучшенными вычислительными возможностями. Как правило, в подключаемых процессорах реализуются специальные функции, такие как обработка сетевых пакетов, обработка мультимедийных данных, криптография и т. д.

Производительность специализированных приложений за счет этих функций может быть повышена в 5–10 раз.

Чтобы повысить производительность в сто, тысячу или миллион раз, необходимо свести воедино многочисленные процессоры и обеспечить их эффективное взаимодействие. Этот принцип реализуется в виде больших мультипроцессорных систем и мультикомпьютеров (кластерных компьютеров). Естественно, объединение тысяч процессоров в единую систему порождает новые проблемы, которые нужно решать.

Наконец, в последнее время появилась возможность интеграции через Интернет целых организаций. В результате формируются слабо связанные распределенные вычислительные сети, или матрицы (grids). Такие системы только начинают развиваться, но их потенциал весьма высок.

Когда два процессора или обрабатывающих элемента находятся рядом и обмениваются большими объемами данных с небольшими задержками, они называются **сильно связанными** (tightly coupled). Соответственно, когда два процессора или обрабатывающих элемента располагаются далеко друг от друга и обмениваются небольшими объемами данных с большими задержками, они называются **слабо связанными** (loosely coupled). В данной главе мы обсудим принципы разработки систем этих форм параллелизма и рассмотрим ряд примеров. Начав с сильно связанных систем, для которых характерен внутрипроцессорный параллелизм, мы постепенно перейдем к слабо связанным системам и в завершающей части главы поговорим о распределенных вычислительных системах. Примерный спектр рассматриваемых тем иллюстрирует рис. 8.1.

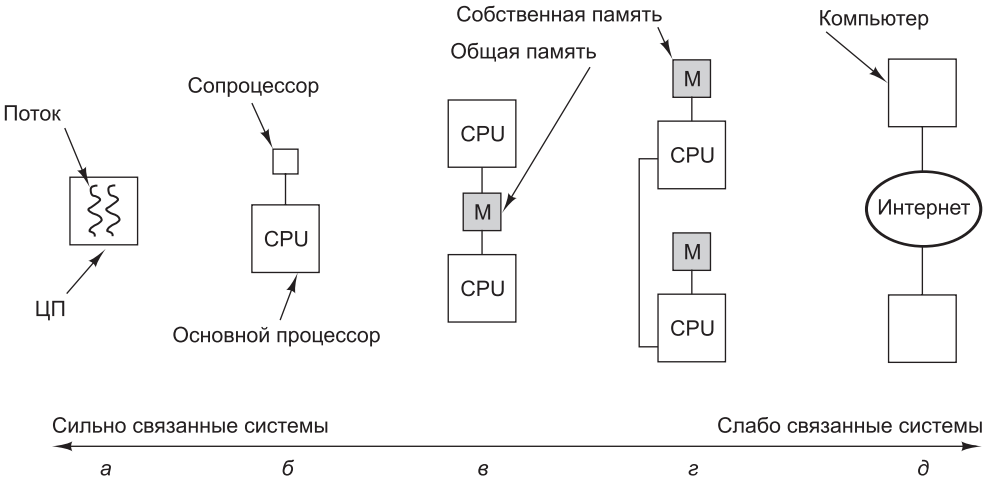


Рис. 8.1. Внутрипроцессорный параллелизм (*а*); сопроцессор (*б*); мультипроцессор (*в*); мультикомпьютер (*г*); слабо связанная распределенная вычислительная система (*д*)

Параллелизм постоянно оказывается темой горячих дискуссий, в связи с чем в этой главе необычно много ссылок — в основном, на недавние работы по заданной теме. Дополнительные ссылки на работы ознакомительного характера можно найти в соответствующем разделе главы 9.

Внутрипроцессорный параллелизм

Один из очевидных способов увеличить производительность микросхемы состоит в том, чтобы заставить ее выполнять больше операций в единицу времени. В этом разделе мы рассмотрим некоторые приемы повышения быстродействия за счет параллелизма на уровне микросхемы, в частности параллелизм на уровне команд, многопоточность и размещение на микросхеме нескольких процессоров. Все эти методики хоть и отличаются друг от друга, в той или иной степени помогают решить задачу. Их роднит базовый принцип — «уплотнение» операций во времени.

Параллелизм на уровне команд

Низкоуровневый параллелизм достигается, в частности, вызовом нескольких команд за один тактовый цикл. Процессоры, в которых реализуется этот принцип, делятся на две категории: суперскалярные и VLIW. Те и другие уже упоминались в предыдущих главах, но сейчас этот материал нелишне повторить.

Схема суперскалярного процессора приведена на рис. 2.5. В наиболее распространенных конфигурациях команда должна быть готова к исполнению в определенной точке конвейера. Суперскалярные процессоры способны за один тактовый цикл вызывать несколько команд. Число фактически вызываемых команд зависит как от конструкции процессора, так и от текущей ситуации. Аппаратные ограничения диктуют максимальное число одновременно вызываемых команд — обычно от двух до шести. К тому же, если для исполнения команды нужен недоступный функциональный блок или еще не полученный результат выполнения другой команды, такая команда не будет вызвана даже при наличии физической возможности.

Другой вид параллелизма на уровне команд реализуется в процессорах **со сверхдлинным командным словом** (Very Long Instruction Word, **VLIW**). В своем первоначальном исполнении VLIW-системы, действительно, отличались длинными словами с командами, обращавшимися к нескольким функциональным блокам. Для примера рассмотрим конвейер, изображенный на рис. 8.2, *а*. Он включает в себя пять функциональных блоков и способен одновременно исполнять две целочисленные операции, одну операцию с плавающей точкой, одну команду загрузки и одну команду сохранения. В одной команде этой VLIW-системы содержится пять кодов операций и пять пар операндов — по одному коду и одной паре на каждый функциональный блок. С учетом того, что код операции занимает 6 бит, регистр — 5 бит, а ячейка памяти — 32 бита, общая длина команды может достигать 134 бита, что, согласитесь, немало.

Однако такое решение было признано неудачным. Дело в том, что не все команды могли обращаться к соответствующим функциональным блокам, в результате в изобилии появлялись бессмысленные пустые операции (рис. 8.2, *б*). В современных VLIW-системах должен быть предусмотрен какой-либо механизм маркировки связок команд, например, для этого может использоваться бит «завершения связки» (рис. 8.2, *в*). Процессор может выбрать и запустить связку целиком. Задача по подготовке связок команд, которые могут выполняться совместно, решается компилятором.

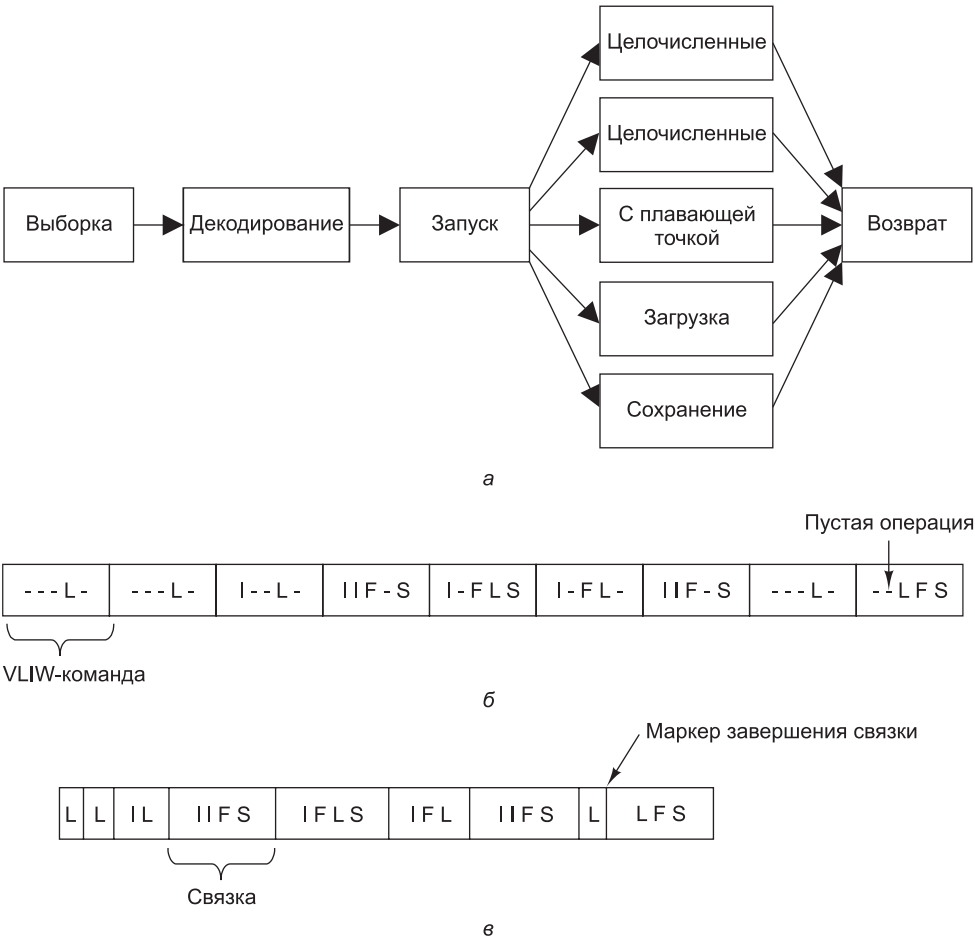


Рис. 8.2. Конвейер процессора (*а*); последовательность VLIW-команд (*б*); поток команд с отмеченными связками (*в*)

Фактически, в VLIW-системах решение проблемы совместимости команд переносится с периода исполнения на стадию компиляции. В результате аппаратное обеспечение упрощается и удешевляется. Кроме того, поскольку в работе компилятора нет жестких временны́х ограничений, связки команд формируются более осмысленно, нежели если бы это происходило в период исполнения. К сожалению, ввести в практику столь радикальные преобразования архитектуры процессора очень сложно, что подтверждается более чем умеренными темпами распространения процессора Itanium.

Нелишне заметить, что параллелизм на уровне команд не является единственно возможной формой низкоуровневого параллелизма. Существует также параллелизм на уровне памяти, предусматривающий одновременное исполнение в памяти множества операций [Chou et al., 2004].

VLIW-процессор TriMedia

В главе 5 на примере процессора Itanium 2 мы уже сталкивались с архитектурой VLIW. Теперь познакомимся с другим VLIW-процессором — **TriMedia** производства компании Philips. TriMedia — это встроенный процессор для устройств обработки изображений, а также аудио- и видеоустройств, таких как CD-, DVD- и MP3-плееры, устройства записи CD и DVD, интерактивные телевизоры, цифровые фотокамеры, видеокамеры и т. д. Учитывая специализацию, нет ничего удивительного в многочисленных отличиях TriMedia от Itanium 2 — универсального процессора для высокопроизводительных серверов.

В состав одной TriMedia-команды может входить до пяти **операций**. В полностью оптимальных условиях за тактовый цикл запускается одна команда и выбирается пять операций. Номинальная тактовая частота процессора составляет 266 или 366 МГц, но так как за один цикл может исполняться до пяти операций, фактическое быстродействие в пять раз больше. При дальнейшем изложении мы будем исходить из характеристик реализации TM3260 процессора. Другие версии TriMedia имеют ряд малозначительных отличий.

Стандартная TriMedia-команда изображена на рис. 8.3. По своему характеру команды варьируются от общераспространенных 8-, 16- и 32-разрядных целочисленных команд до команд с плавающей точкой стандарта IEEE 754 и команд параллельной обработки мультимедиа. Исполнение пяти операций за цикл и наличие команд параллельной обработки мультимедийных данных позволяют процессору TriMedia программно декодировать потоковое цифровое видео, поступающее с видеокамеры, сохраняя исходные размер и частоту кадров.

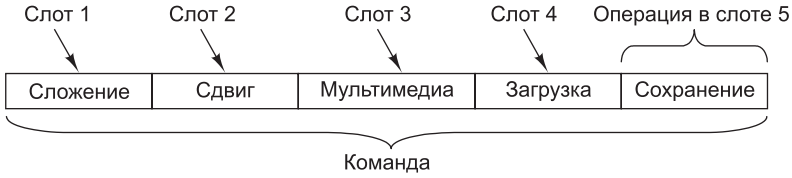


Рис. 8.3. Стандартная TriMedia-команда с пятью операциями

В TriMedia используется память с байтовой организацией, а регистры ввода-вывода отображаются на пространство памяти. Полуслова (16 бит) и полные слова (32 бит) выравниваются по естественным границам. Порядок следования байтов может быть как прямым, так и обратным — в зависимости от бита слова состояния программы, выставляемого операционной системой. Этот бит определяет механизм передачи данных операций загрузки и сохранения между памятью и регистрами. В процессоре предусмотрена разделенная 8-входовая ассоциативная кэш-память с одинаковой длиной строки (64 байт) в кэшах команд и данных. Емкость кэша команд составляет 64 Кбайт; емкость кэша данных — 16 Кбайт.

Существует 128 универсальных 32-разрядных регистров. Значения регистров R0 и R1 равны аппаратным нулю и единице соответственно. Остальные 126 регистров функционально эквивалентны и могут применяться для любых целей. Кроме того, предусмотрены четыре специализированных 32-разрядных регистра: счетчик команд, регистр слова состояния программы и два регистра, связанные с прерываниями. Наконец, один 64-разрядный регистр подсчитывает число циклов процессора с момента последнего сброса. При тактовой частоте 300 МГц полный цикл счетчика составляет 2000 лет.

В процессоре Trimedia TM3260 есть 11 функциональных блоков, предназначенных для выполнения арифметических, логических и управляющих операций (также есть блок управления кэшем, но его мы рассматривать не будем). Все они перечислены в табл. 8.1. В первых двух столбцах приводятся имя блока и краткое описание выполняемых им функций. В третьем столбце указывается число аппаратных копий блока. Четвертый столбец содержит значение ожидания (точнее, число циклов) до завершения операции. В этом контексте нелишне отметить, что все функциональные блоки, за исключением блока извлечения квадратного корня и деления чисел с плавающей точкой, конвейеризированы. Хотя ожидание говорит о том, сколько нужно ждать до завершения операции, не следует забывать, что в каждом новом цикле можно начинать новые операции. Так, в каждой из трех последовательных команд может содержаться по две операции загрузки, а значит, одновременно на разных этапах исполнения может находиться шесть операций загрузки.

Таблица 8.1. Функциональные блоки TM3260 с указанием их количества, задержки и соответствия командным слотам

Блок	Описание	#	Ожидание	1	2	3	4	5
Операции с константами	Операции с непосредственной адресацией	5	1	Да	Да	Да	Да	Да
АЛУ целочисленных операций	32-разрядные арифметические и логические операции	5	1	Да	Да	Да	Да	Да
Сдвиги	Многоразрядные сдвиги	2	1	Да	Да	Да	Да	Да
Загрузка и сохранение	Обращения к памяти	2	3				Да	Да
Умножение целых чисел и чисел с плавающей точкой	32-разрядные операции умножения целых чисел и чисел с плавающей точкой	2	3		Да	Да		
АЛУ операций с плавающей точкой	Арифметические операции с плавающей точкой	2	3	Да			Да	
Сравнение чисел с плавающей точкой	Операции сравнения чисел с плавающей точкой	1	1			Да		
Извлечение квадратного корня и деления чисел с плавающей точкой	Деление и извлечение квадратного корня для чисел с плавающей точкой	1	17		Да			
Переходы	Управление потоком операций	3	3		Да	Да	Да	
АЛУ цифровой обработки сигналов	Арифметические операции с мультимедийными данными (по два 16-разрядных или по четыре 8-разрядных слова)	2	3	Да		Да		Да
Умножитель для цифровой обработки сигналов	Умножение мультимедийных данных (по два 16-разрядных или по четыре 8-разрядных слова)	2	3		Да	Да		

Наконец, в последних шести столбцах определяется соответствие команд функциональным блокам. К примеру, операции сравнения чисел с плавающей точкой могут проводиться только в третьем командном слоте.

Функциональный блок операций с константами применяется при выполнении операций с непосредственной адресацией, например при загрузке числа из поля операции в регистр. АЛУ для исполнения целочисленных операций осуществляет сложение, вычитание, стандартные логические операции, а также операции упаковки и распаковки. Блок операций сдвига может выполнять сдвиги регистра на указанное число битов в обоих направлениях.

Блок загрузки и сохранения считывает слова из памяти в регистры и записывает их обратно. По большому счету, TriMedia — это RISC-процессор с расширенной функциональностью, поэтому обычные операции выполняются с регистрами, а обращения к памяти осуществляются при помощи функционального блока загрузки и сохранения. Передача может осуществляться по 8, 16 или 32 бит. При исполнении арифметических и логических команд обращение к памяти не осуществляется.

Блок умножения выполняет операции как с целыми числами, так и с числами с плавающей точкой. Следующие три блока ответственны за сложение и вычитание чисел с плавающей точкой, сравнение, извлечение квадратных корней и деление.

Операции перехода исполняются функциональным блоком переходов. За переходом следует фиксированная задержка величиной в три цикла, за время которых всегда исполняются три команды (то есть до 15 операций). Так происходит даже при безусловных переходах.

Наконец, мы дошли до двух блоков, предназначенных для выполнения специальных мультимедийных операций. Фактически мультимедийные операции выполняет обработчик цифровых сигналов (Digital Signal Processor, DSP). Следует сразу заметить, что в отличие от целочисленных операций, основывающихся на арифметике с дополнением до двух, в мультимедийных операциях используется **арифметика с насыщением** (saturated arithmetic). Если результат операции нельзя выразить из-за переполнения, вместо вызова исключения или возвращения в качестве результата «мусора» подставляется ближайшее корректное число. К примеру, применительно к 8-разрядным числам без знака в результате сложения 130 и 130 может получиться 255.

Так как некоторые операции и командные слоты несовместимы, зачастую в команду включается меньше пяти операций. Если тот или иной слот не используется, для минимизации потребляемого пространства он подлежит сжатию. Присутствующие в команде операции могут занимать 26, 34 или 42 бита. В зависимости от числа фактически содержащихся в TriMedia-команде операций, ее размер составляет от 2 до 28 байт (включая служебные данные фиксированного размера).

Проверка входящих в TriMedia-команду операций на совместимость в период исполнения не проводится. Поэтому операции исполняются даже в случае их несовместимости, что порождает неверный результат. Решение отказаться от проверок было принято разработчиками для экономии времени и транзисторов. В процессорах Core i7 проверка совместимости суперскалярных операций проводится, однако в результате усложняется решение, растут временны́е затраты и увеличивается число применяемых транзисторов. В TriMedia задача планиро-

вания передается компилятору, который без лишних временны́х ограничений может оптимизировать размещение операций в словах команд. Однако если для исполнения операции требуется недоступный функциональный блок, вся команда вынуждена ждать вплоть до того момента, как он вновь станет доступным.

Как и в Itanium 2, в TriMedia операции прогнозируются. Каждая операция (за двумя незначительными исключениями) задает регистр, подлежащий проверке перед исполнением этой операции. Если младший бит этого регистра установлен, операция исполняется; в противном случае она пропускается. Каждая из пяти (или менее) операций прогнозируется индивидуально. Вот пример спрогнозированной операции:

```
IF R2 IADD R4, R5 -> R8
```

Здесь проверяется регистр R2, и если значение его младшего бита равно единице, содержимое регистров R4 и R5 складывается и сохраняется в R8. Операцию можно сделать безусловной, если в качестве предикатного регистра использовать R1 (его значение всегда равно 1). Регистр R0 (аппаратный ноль) делает операцию пустой.

Мультимедийные операции в TriMedia подразделяются на 15 групп, перечисленных в табл. 8.2. Во многих из этих операций применяется отсечение — методика, в которой операнд «загоняется» в определенный диапазон, исходя из минимальных или максимальных значений операндов вне этого диапазона. Отсечение осуществимо в отношении 8-, 16- и 32-разрядных операндов. К примеру, в результате отсечения значений от 40 до 340 по диапазону от 0 до 255 остаются значения от 40 до 255. Операции отсечения проводятся в группе отсечения.

Таблица 8.2. Основные группы специализированных операций в TriMedia

Группа	Описание
Отсечение	Отсечение четырех байт или двух полуслов
Получение абсолютного значения (DSP)	Получение абсолютного значения, получение знака, отсечение
Сложение (DSP)	Сложение значений с учетом знака и отсечением
Вычитание (DSP)	Вычитание значений с учетом знака и отсечением
Умножение (DSP)	Перемножение значений с учетом знака и отсечением
Получение минимума и максимума	Получение минимальной или максимальной из четырех пар байтов
Сравнение	Побайтовое сравнение двух регистров
Сдвиг	Сдвиг пары 16-разрядных операндов
Сумма произведений	Суммирование с учетом знака 8- или 16-разрядных произведений
Слияние, упаковка, перестановка	Манипулирование байтами и полусловами
Побайтовое квадратичное усреднение	Побайтовое квадратичное усреднение без учета знака

продолжение ⇩

Таблица 8.2 *(продолжение)*

Группа	Описание
Побайтовое усреднение	Побайтовое усреднение из четырех элементов без учета знака
Побайтовое умножение	Умножение 8-разрядных значений без учета знака
Оценка движения	Суммирование без учета знака абсолютных значений 8-разрядных разностей со знаком
Разное	Другие арифметические операции

Следующие пять групп в табл. 8.2 объединяют операции с операндами различных размеров, предусматривающие отсечение результатов по определенному диапазону. Операции группы получения минимума и максимума анализируют два регистра и находят для каждого байта минимальное и максимальное значения. Аналогичным образом, в группе сравнения два регистра рассматриваются как четыре пары байтов, каждая из которых подлежит сравнению с остальными.

Мультимедийные операции довольно редко выполняются с 32-разрядными целыми числами. Связано это с тем, что изображения обычно строятся в цветовой модели RGB (Red, Green, Blue — красный, зеленый, синий) с 8-разрядными значениями пикселов красного, зеленого и синего цветов. При обработке (например, сжатии) изображения оно выражается тремя компонентами, по одному на каждый цвет (в пространстве RGB), или в логически эквивалентной форме (в пространстве YUV, которое мы обсудим далее). В любом случае, основной объем вычислений проводится для прямоугольных матриц 8-разрядных целых чисел без знака.

Для эффективной обработки таких матриц в TriMedia предусмотрены многочисленные специализированные операции. В качестве простого примера рассмотрим верхний левый угол матрицы 8-разрядных значений, сохраненной в памяти с прямым порядком следования байтов (рис. 8.4, *а*). Блок 4 × 4 в этом углу содержит 16 8-разрядных значений от *A* до *P*. Предположим, что в результате транспонирования изображения получилась матрица, изображенная на рис. 8.4, *б*. Как достигается этот результат?

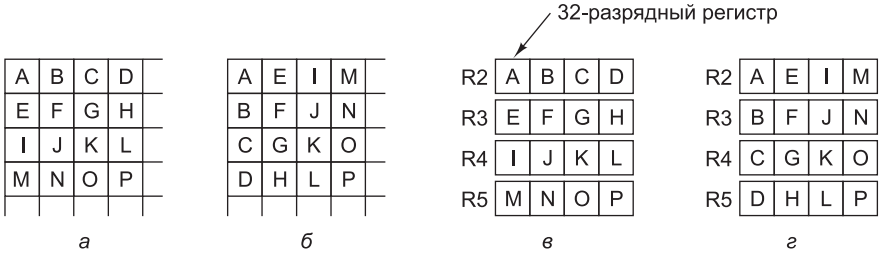


Рис. 8.4. Матрица 8-разрядных элементов (*а*); транспонированная матрица (*б*); исходная матрица, перенесенная в четыре регистра (*в*); транспонированная матрица в четырех регистрах (*г*)

Транспонирование можно провести за 12 операций, каждая из которых загружает байты в новые регистры, после которых должно быть выполнено еще 12 операций, помещающих эти байты по назначению (отметьте, что четыре диагональных байта при транспонировании не перемещаются). Проблема в том, что эта схема требует 24 длинных и длительных операций, обращающихся к памяти.

Есть и другой способ. Сначала выполняются четыре операции, каждая из которых загружает одно слово в четыре разных регистра — от R2 до R5 (как показано на рис. 8.4, *в*). Затем с помощью операций маскирования и сдвига четыре получившихся слова объединяются, и формируется желаемый результат (рис. 8.4, *г*). В конце слова сохраняются в памяти. Несмотря на значительное сокращение числа обращений к памяти (с 24 до 8), эффективность этого метода не высока из-за маскирования и сдвига — для извлечения и размещения всех байтов в нужных местах требуется слишком много операций.

В TriMedia реализован более удачный метод. Сначала четыре слова размещаются в регистрах. При этом результат формируется не маскированием и сдвигами, а специализированными операциями для извлечения и размещения байтов в регистрах. Таким образом, для транспонирования изображения достаточно восьми специальных мультимедийных операций и такого же числа обращений к памяти. Код начинается с двух операций загрузки в сегменты 4 и 5 для размещения слов в регистры R2 и R3, за которыми следуют аналогичные операции для загрузки в регистры R4 и R5.

Команды, в составе которых находятся эти операции, могут задействовать сегменты 1, 2 и 3 для любых других целей. После загрузки всех слов 8 специальных мультимедийных операций вместе с двумя операциями сохранения можно упаковать в две команды, формирующие выходные данные. В конечном счете, требуются всего 6 команд, причем 14 из 30 слотов остаются доступными для других операций, а значит, при решении поставленной задачи число слотов соответствует примерно трем командам. Другие мультимедийные операции столько же эффективны. Благодаря этим операциям, а также разделению команды на пять слотов процессор TriMedia оказывается высокоэффективным инструментом обработки мультимедийных данных.

Внутрипроцессорная многопоточность

Для всех современных конвейеризованных процессоров характерна одна и та же проблема — если при запросе к памяти слово не обнаруживается в кэшах первого и второго уровней, на загрузку этого слова в кэш уходит длительное время, в течение которого конвейер простаивает. Одна из методик решения этой проблемы называется **внутрипроцессорной многопоточностью** (on-chip multithreading). Она позволяет процессору одновременно управлять несколькими программными потоками и тем самым маскировать простои. Вкратце принцип многопоточности можно изложить так: если программный поток 1 блокируется, процессор может обеспечить полную загрузку аппаратуры, запустив программный поток 2.

Основополагающая идея проста, реализуется она разными способами, которые мы и рассмотрим. Первый из них, называемый **мелкомодульной многопоточностью** (fine-grained multithreading), применительно к процессору, способному вызывать одну команду за такт, иллюстрирует рис. 8.5. На рис. 8.5, *а–в* изображено три программных потока (*A*, *B*, *C*), соответствующих 12 машинным циклам. В ходе первого цикла поток *A* исполняет команду *A*1. Поскольку эта команда завершается за один цикл, при наступлении второго цикла запускается команда *A*2. Ее обращение в кэш первого уровня оказывается неудачным, поэтому до извлечения нужного слова из кэша второго уровня проходит два цикла. Исполнение

потока продолжается в цикле 5. Как показано на рисунке, потоки *B* и *C* также регулярно простаивают. В рамках такого решения вызов последующей команды до завершения предыдущей не осуществляется. Точнее, при наличии сложного счетчика обращений в некоторых случаях это допустимо, но такую возможность мы для простоты исключаем.

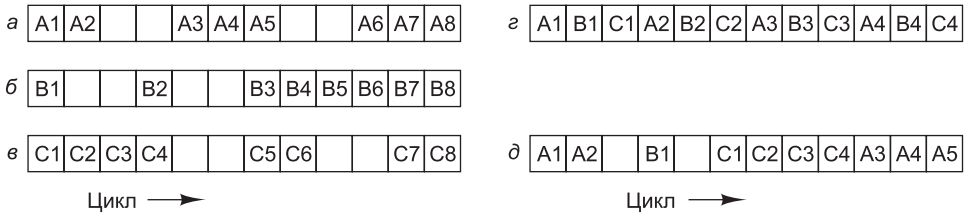


Рис. 8.5. Три программных потока. Пустые квадраты означают простой в ожидании данных из памяти (*а–в*); мелкомодульная многопоточность (*г*); крупномодульная многопоточность (*д*)

При мелкомодульной многопоточности простой маскируется путем исполнения потоков «по кругу», то есть в смежных циклах запускаются разные потоки (рис. 8.5, *г*). К моменту наступления цикла 4 обращение к памяти, инициированное командой *A*1, завершается, поэтому даже если команде *A*2 нужен результат команды *A*1, она запускается. В таком случае максимальная продолжительность простоя составляет два цикла, то есть при наличии трех программных потоков простаивающая операция все равно завершается вовремя. При простое в 4 цикла для беспрерывной работы понадобилось бы 4 программных потока и т. д.

Поскольку разные программные потоки никак друг с другом не связаны, каждому из них нужен свой набор регистров. Он должен быть указан для каждой вызываемой команды, и тогда аппаратное обеспечение будет знать, к какому набору регистров при необходимости нужно обращаться. Следовательно, максимальное число одновременно исполняемых программных потоков определяется в период разработки микросхемы.

Обращениями к памяти причины простоя не ограничиваются. Иногда для исполнения следующей команды требуется результат предыдущей команды, который еще не вычислен. В других случаях команда вызвана быть не может, так как она следует за условным переходом, направление которого еще неизвестно. Общее правило формулируется так: если в конвейере k ступеней, но по кругу можно запустить, по меньшей мере, k программных потоков, то в одном потоке в любой отдельно взятый момент не может исполняться более одной команды, поэтому конфликты между ними исключены. В такой ситуации процессор может работать на полной скорости, без простоя.

Естественно, далеко не всегда число доступных потоков равно числу ступеней конвейера, поэтому некоторые разработчики предпочитают методику, называемую **крупномодульной многопоточностью** (coarse-grained multithreading), которую иллюстрирует рис. 8.5, *д*. В данном случае программный поток *A* продолжает исполняться последовательно, вплоть до простоя. При этом теряется один цикл. Далее происходит переключение на первую команду программного потока *B* (*B*1). Так как эта команда сразу переходит в состояние простоя, в цикле 6 исполняется уже команда *C*1. Так как каждый раз при простое команды теряется

один цикл, по своей эффективности крупномодульная многопоточность, казалось бы, уступает мелкомодульной, однако у нее есть одно существенное преимущество — за счет меньшего числа программных потоков значительно сокращается расход ресурсов процессора. При недостаточном количестве активных потоков эта методика оптимальна.

Судя по нашему описанию, при крупномодульной многопоточности просто выполняется переключение между потоками, однако это не единственный возможный вариант действий. Также возможно немедленное переключение с команд, которые потенциально способны вызвать простой (например, загрузка, сохранение и переходы) без выяснения, действительно ли намечается простой. Эта стратегия позволяет переключаться раньше обычного (сразу после декодирования команды) и исключает бесконечные циклы. Иными словами, исполнение продолжается до того момента, пока не обнаружится возможность возникновения проблемы, после чего следует переключение. Такие частые переключения роднят крупномодульную многопоточность с мелкомодульной.

Независимо от используемого варианта многопоточности необходимо как-то отслеживать принадлежность каждой операции к тому или иному программному потоку. В рамках мелкомодульной многопоточности каждой операции присваивается идентификатор потока, поэтому при перемещениях по конвейеру ее принадлежность не вызывает сомнений. Крупномодульная многопоточность предусматривает возможность очистки конвейера перед запуском каждого последующего потока. В таком случае четко определяется идентичность потока, исполняемого в данный момент. Естественно, данная методика эффективна только в том случае, если паузы между переключениями значительно больше времени, необходимого для очистки конвейера.

Все сказанное относится к процессорам, способным вызывать не более одной команды за тактовый цикл. Однако мы знаем, что для современных процессоров это ограничение не актуально. Применительно к изображению на рис. 8.6 мы допускаем, что процессор может вызывать по две команды за цикл, однако утверждение о невозможности запуска последующих команд в случае простоя предыдущей остается в силе. Рисунок 8.6, *а* иллюстрирует механизм мелкомодульной многопоточности в сдвоенном суперскалярном процессоре. Как видно, в потоке *A* первые две команды запускаются во время первого цикла, однако в потоке *B* во втором цикле запускается только одна команда.

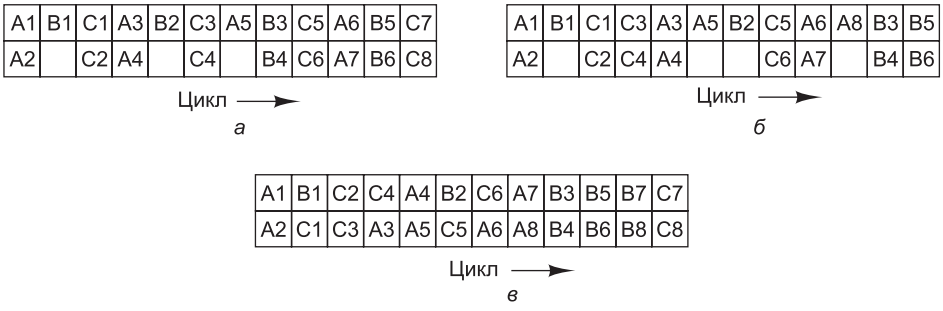


Рис. 8.6. Многопоточность в сдвоенном процессоре: мелкомодульная многопоточность (*а*); крупномодульная многопоточность (*б*); синхронная многопоточность (*в*)

На рис. 8.6, *б* изображена реализация крупномодульной многопоточности в сдвоенном процессоре со статическим планировщиком, который исключает бесконечные циклы при простое команд. Здесь программные потоки исполняются по очереди, процессор вызывает по две команды в каждом потоке, пока не обнаруживает простоя; в следующем такте после простоя начинается исполнение следующего потока.

В суперскалярных процессорах есть еще один способ организации многопоточности — так называемая **синхронная многопоточность** (simultaneous multithreading), которую иллюстрирует рис. 8.6, *в*. Эта методика представляет собой усовершенствованный вариант крупномодульной многопоточности, где каждый программный поток может запускать по две команды за такт, однако в случае простоя с целью обеспечения полной загрузки процессора запускаются команды следующего потока. При синхронной многопоточности полностью загружаются все функциональные блоки. В случае невозможности запуска команды из-за занятости функционального блока выбирается команда из другого потока. На рисунке предполагается, что в цикле 11 простаивает команда *B*8, поэтому в цикле 12 запускается команда *C*7.

Дополнительные сведения о многопоточности можно почерпнуть в [Gebhart et al., 2011; Wing-kei et al., 2011].

Многопоточность в Core i7

Разобравшись с теорией многопоточности, рассмотрим практический пример — Core i7. В начале 2000-х годов такие процессоры, как Pentium 4, уже не обеспечивали прироста производительности, необходимого Intel для поддержания продаж на нужном уровне. Уже после того, как процессор Pentium 4 пошел в производство, инженеры Intel продолжали работу над повышением его быстродействия без внесения изменений в программный интерфейс. Быстро наметились пять простейших способов.

1. Повышение тактовой частоты.
2. Размещение на одной микросхеме двух процессоров.
3. Введение новых функциональных блоков.
4. Удлинение конвейера.
5. Использование многопоточности.

Самый очевидный способ повышения быстродействия заключается в том, чтобы повысить тактовую частоту, не меняя другие параметры. Как правило, каждая последующая модель процессора имеет несколько более высокую тактовую частоту, чем предыдущая. К сожалению, при прямолинейном повышении тактовой частоты разработчики сталкиваются с двумя проблемами: увеличением энергопотребления (что актуально для портативных компьютеров и других вычислительных устройств, работающих на аккумуляторах) и перегревом (что требует создания более эффективных теплоотводов).

Второй способ — размещение на микросхеме двух процессоров — сравнительно прост, но он сопряжен с удвоением площади, занимаемой микросхемой. Если каждый процессор снабжается собственной кэш-памятью, количество микросхем на пластине уменьшается вдвое, но это также означает удвоение затрат на производство. Если для обоих процессоров предусматривается общая кэш-память,

значительного увеличения занимаемой площади удается избежать, однако в этом случае возникает другая проблема — объем кэш-памяти в пересчете на каждый процессор уменьшается вдвое, а это неизбежно сказывается на производительности. Кроме того, если высокопроизводительные серверные приложения способны полностью задействовать ресурсы нескольких процессоров, то в обычных настольных программах внутренний параллелизм развит в значительно меньшей степени.

Добавить новые функциональные блоки несложно, но здесь важно соблюсти баланс. Какой смысл в десятке блоков АЛУ, если микросхема не может выдавать команды на конвейер с такой скоростью, которая позволяет загрузить все эти блоки?

Конвейер с увеличенным числом ступеней, способный разделять задачи на более мелкие сегменты и обрабатывать их за короткие периоды времени, с одной стороны, повышает производительность, с другой — усиливает негативные последствия неверного прогнозирования переходов, кэш-промахов, прерываний и других событий, нарушающих нормальный ход обработки команд в процессоре. Кроме того, чтобы полностью реализовать возможности расширенного конвейера, необходимо повысить тактовую частоту, а это, как мы знаем, приводит к повышенным энергопотреблению и теплоотдаче.

Наконец, можно реализовать многопоточность. Преимущество этой технологии состоит во введении дополнительного программного потока, позволяющего ввести в действие те аппаратные ресурсы, которые в противном случае простаивали бы. По результатам экспериментальных исследований разработчики Intel выяснили, что увеличение площади микросхемы на 5 % при реализации многопоточности для многих приложений дает прирост производительности на 25 %. Первым процессором Intel с поддержкой многопоточности стал Xeon 2002 года. Впоследствии, начиная с частоты 3,06 ГГц, многопоточность была внедрена в линейку Pentium 4 (включая Core i7). Intel называет реализацию многопоточности в своих процессорах **гиперпоточностью** (hyperthreading).

Основной принцип гиперпоточности — одновременное исполнение двух программных потоков (или процессов — процессор не отличает процессы от программных потоков). Операционная система рассматривает гиперпоточный процессор Core i7 как двухпроцессорный комплекс с общими кэшами и основной памятью. Планирование операционная система выполняет для каждого программного потока отдельно. Таким образом, в одно и то же время могут исполняться два приложения. К примеру, почтовый демон может отправлять или принимать сообщения в фоновом режиме, пока пользователь взаимодействует с интерактивным приложением — то есть демон и пользовательская программа исполняются одновременно, как будто системе доступно два процессора.

Прикладные программы, предусматривающие возможность исполнения в виде нескольких программных потоков, могут задействовать оба «виртуальных процессора». Например, программы редактирования видеоданных обычно позволяют пользователям применять фильтры ко всем кадрам. Такие фильтры корректируют яркость, контраст, цветовой баланс и другие свойства кадров. В такой ситуации программа может назначить один виртуальный процессор для обработки четных кадров, а другой — для обработки нечетных. При этом два процессора будут работать совершенно независимо друг от друга.

Поскольку программные потоки обращаются к одним и тем же аппаратным ресурсам, необходима координация этих потоков. В контексте гиперпоточности разработчики Intel выделили четыре полезных стратегии управления совместным потреблением ресурсов: дублирование ресурсов, а также жесткое, пороговое и полное разделение ресурсов. Рассмотрим эти стратегии.

Начнем с **дублирования ресурсов** (resource duplication). Как известно, некоторые ресурсы с целью организации программных потоков дублируются. Например, так как каждому программному потоку требуется индивидуальное управление, нужен второй счетчик команд. Кроме того, необходимо ввести вторую таблицу отображения архитектурных регистров (EAX, EBX и т. д.) на физические регистры; аналогичным образом дублируется контроллер прерываний, поскольку обработка прерываний для каждого потока производится индивидуально.

Далее следует методика **жесткого разделения ресурсов** (partitioned resource sharing) между программными потоками. К примеру, если в процессоре предусмотрена очередь между двумя функциональными ступенями конвейера, то половину слотов можно отдавать потоку 1, другую половину — потоку 2. Разделение ресурсов легко реализуется, не ведет к дисбалансу и обеспечивает полную независимость программных потоков друг от друга. При полном разделении всех ресурсов один процессор фактически превращается в два. С другой стороны, может сложиться такая ситуация, при которой один программный поток не использует ресурсы, которые могли бы пригодиться второму потоку, но в отношении которых у него нет полномочий доступа. В результате ресурсы, которые в иной ситуации могли бы быть задействованы, простаивают.

Противоположность жесткого разделения — **полное разделение ресурсов** (full resource sharing). В этой схеме к нужным ресурсам может получить доступ любой программный поток, а обслуживаются они в порядке поступления обращений. Рассмотрим ситуацию, в которой быстрый поток, состоящий преимущественно из операций сложения и вычитания, сосуществует с медленным потоком, реализующим операции умножения и деления. Если команды вызываются из памяти быстрее, чем выполняются операции умножения и деления, число команд, вызванных в рамках медленного потока и поставленных в очередь на конвейер, будет постепенно расти.

В конечном итоге эти команды заполнят очередь, в результате быстрый поток из-за нехватки места в ней остановится. Полное разделение ресурсов решает проблему неоптимального расходования общих ресурсов, но создает дисбаланс их потребления — один поток может замедлить или остановить другой.

Промежуточная схема реализуется в рамках **порогового разделения ресурсов** (threshold resource sharing). Согласно этой схеме любой программный поток может динамически получать определенный (ограниченный) объем ресурсов. Применительно к реплицированным ресурсам этот подход обеспечивает гибкость без угрозы простоя одного из программных потоков из-за невозможности получения ресурсов. Если, к примеру, запретить каждому из потоков занимать больше 3/4 очереди команд, повышенное потребление ресурсов медленным потоком не помешает исполнению быстрого. Модель гиперпоточности Core i7 объединяет разные стратегии разделения ресурсов. Таким образом, предпринимается попытка решить все проблемы, связанные с каждой стратегией. Дублирование реализуется в отношении ресурсов, доступ к которым постоянно требуется обоим программ-

ным потокам (в частности, в отношении счетчика команд, таблицы отображения регистров и контроллера прерываний). Дублирование этих ресурсов увеличивает площадь микросхемы всего лишь на 5 % — согласитесь, вполне разумная плата за многопоточность. Ресурсы, доступные в таком объеме, что практически исключается вероятность их захвата одним потоком (например, строки кэша), распределяются динамически. Доступ к ресурсам, контролирующим работу конвейера (в частности, его многочисленные очереди), разделяется — каждому программному потоку отдается половина слотов. Главный конвейер архитектуры Sandy Bridge, реализованной в Core i7, изображен на рис. 8.7; белые и серые области на этой иллюстрации обозначают механизм распределения ресурсов между белым и серым программными потоками.

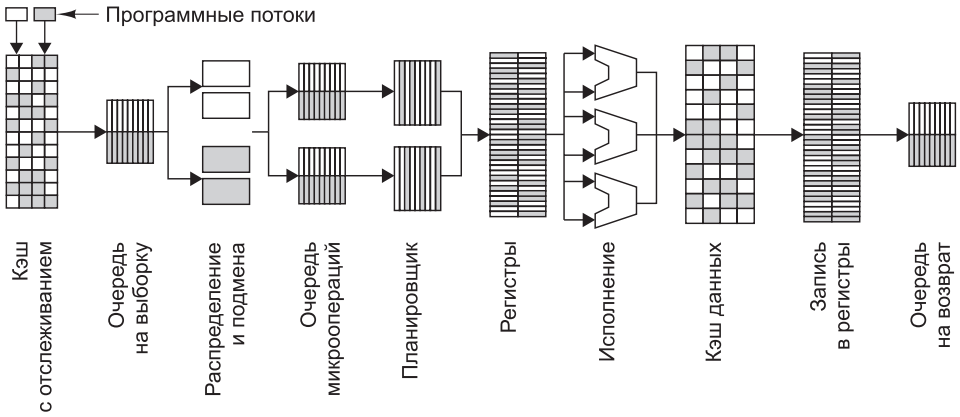


Рис. 8.7. Разделение ресурсов между программными потоками в микроархитектуре Core i7

Как видим, все очереди на этой иллюстрации разделены — каждому программному потоку выделяется по половине слотов. Ни один из программных потоков не может ограничить работу другого. Блок распределения и подмены также разделяется. Ресурсы планировщика разделяются динамически, но на основе некоего порогового значения — таким образом, ни один из потоков не может занять все слоты очереди. Для всех остальных ступеней конвейера имеет место полное разделение.

Впрочем, с многопоточностью не все так просто. Даже у такой прогрессивной методики есть недостатки. Жесткое разделение ресурсов не связано с серьезными издержками, а вот динамическое разделение, в особенности с учетом пороговых величин, требует отслеживать потребление ресурсов на этапе исполнения. Кроме того, в некоторых случаях программы значительно лучше работают без многопоточности, чем с ней. Предположим, к примеру, что при наличии двух программных потоков для нормального функционирования каждому из них требуется 3/4 кэша. Если бы они исполнялись поочередно, каждый показал бы достаточную эффективность при небольшом количестве кэш-промахов (как известно, связанных с дополнительными издержками). В случае параллельного исполнения кэш-промахов у каждого было бы значительно больше, и конечный результат оказался бы хуже, чем без многопоточности.

Дополнительные сведения о механизме многопоточности в процессорах Intel можно почерпнуть в [Gerber and Binstock, 2004; Gepner et al., 2011].

Однокристальные мультипроцессоры

Многопоточность позволяет добиться существенного повышения производительности при разумных затратах, однако некоторым приложениям нужна значительно большая производительность, и многопоточностью тут уже не обойтись. Для таких приложений существуют мультипроцессорные схемы. Микросхемы, на которых устанавливается два или более процессоров, применяются преимущественно в профессиональных серверах и бытовой электронике. Обо всем этом мы сейчас поговорим.

Гомогенные однокристальные мультипроцессоры

Благодаря развитию технологии СБИС (сверхбольшая интегральная схема), в настоящее время на один кристалл можно установить два или более мощных процессоров. Поскольку такие процессоры всегда обращаются к одним и тем же модулям памяти (кэшу первого и второго уровней и основной памяти), они, как мы выяснили в главе 2, считаются единым мультипроцессором. Как правило, они устанавливаются в крупных фермах веб-серверов. При совместном размещении двух процессоров, разделении ресурсов памяти, дисковых и сетевых интерфейсов производительность сервера во многих случаях можно удвоить, причем расходы на это возрастут в значительно меньшей степени (так как даже если мультипроцессор будет стоить в два раза больше обычного процессора, не стоит забывать, что его цена составляет лишь небольшую часть общей стоимости системы).

Для малых однокристальных мультипроцессоров имеются два типовых проектных решения. В первом из них (рис. 8.8, *а*) присутствует одна микросхема и два конвейера — таким образом, скорость исполнения команд теоретически удваивается. Во втором решении (рис. 8.8, *б*) в микросхеме предусмотрено два независимых ядра, каждое из которых содержит полноценный процессор. **Ядром** называется большая микросхема (например, процессор, контроллер ввода-вывода или кэш-память), которая размещается на микросхеме в виде модуля, как правило, вместе с несколькими другими ядрами.

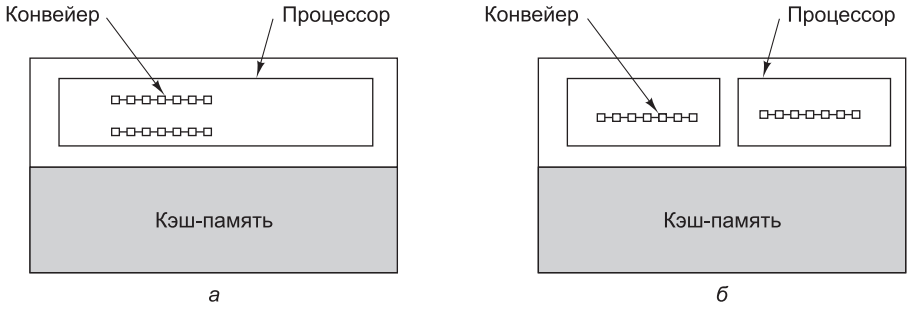


Рис. 8.8. Однокристальные мультипроцессоры: микросхема с двумя конвейерами (*а*); микросхема с двумя ядрами (*б*)

Первое решение допускает совместное потребление ресурсов (таких как функциональные блоки) процессорами; иными словами, каждый процессор может обращаться к ресурсам, не затребованным другим процессором. Второе решение требует изменения конструкции микросхемы и не предусматривает более двух процессоров. Здесь следует заметить, что разместить на одной микросхеме два или несколько ядер процессоров не так уж сложно.

Далее по мере изложения мы еще вернемся к теме мультипроцессоров. В этой главе мы в основном рассматриваем темы, относящиеся к мультипроцессорам на основе однопроцессорных микросхем, но многие из них применимы и к микросхемам с несколькими процессорами.

Однокристальный мультипроцессор Core i7

Core i7 — однокристальный мультипроцессор, производимый с четырьмя и более ядрами на одной кремниевой подложке. Высокоуровневая структура процессора Core i7 изображена на рис. 8.9.

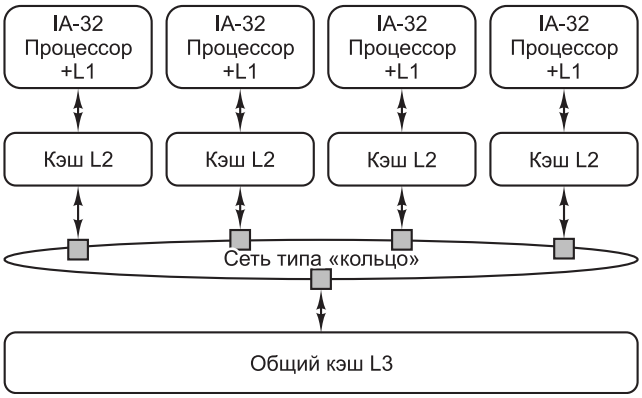


Рис. 8.9. Архитектура однокристального мультипроцессора Core i7

Каждый процессор Core i7 имеет собственные кэши команд и данных первого уровня, а также свой собственный объединенный кэш L2. Процессоры связаны с собственными кэшами по выделенным каналам «точка-точка». На следующем уровне иерархии памяти находится общий объединенный кэш данных L3.

Кэши L2 связываются с общим кэшем L3 по сети типа «кольцо». Запрос, входящий в такую сеть, передается следующему узлу сети. Там узел проверяет, не достиг ли запрос своего целевого узла. Процесс передачи продолжается до тех пор, пока не будет найден целевой узел, или запрос не вернется к источнику (это будет означать, что целевой узел не существует). Основное преимущество кольцевой сети — недорогая реализация при высокой пропускной способности, с дополнительными задержками на перемещение запроса от узла к узлу. Кольцевая сеть Core i7 имеет два основных предназначения: во-первых, она обеспечивает возможность перемещения запросов к памяти и запросов ввода-вывода между кэшами и процессорами, а во-вторых — реализует проверки, гарантирующие, что все процессоры всегда согласованно воспринимают состояние памяти. Проверки согласования будут описаны позднее в этой главе.

Гетерогенные однокристальные мультипроцессоры

Помимо серверов, однокристальные мультипроцессоры используются во встроенных системах — по большей части в аудиовизуальной бытовой электронике: телевизорах, DVD-плеерах, видеокамерах, игровых приставках, сотовых телефонах и т. д. Подобные системы отличаются повышенными требованиями к производительности и жесткими ограничениями. Несмотря на различия во внешнем виде, по существу, эти устройства представляют собой миниатюрные компьюте-

ры с одним или несколькими процессорами, модулями памяти, контроллерами и специализированными устройствами ввода-вывода. К примеру, сотовый телефон — это компьютер, оснащенный процессором, памятью, компактной клавиатурой, микрофоном, системой воспроизведения звукового сигнала и беспроводным сетевым адаптером.

Рассмотрим для примера портативный DVD-плеер. Размещенный в нем компьютер призван выполнять несколько функций:

1. Управление дешевым и ненадежным сервоприводом, регулирующим положение головки.
2. Преобразование аналогового сигнала в цифровую форму.
3. Коррекция ошибок.
4. Дешифрирование и управление правами.
5. Декомпрессия данных видеоформата MPEG-2.
6. Декомпрессия звуковых данных.
7. Кодирование выходных данных для воспроизведения в телевизионных системах NTSC, PAL или SECAM.

Все эти функции должны выполняться в реальном времени с соблюдением жестких ограничений по качеству обслуживания, энергопотреблению, отводу тепла, размерам, весу и стоимости.

Данные на дисках CD, DVD и Blu-ray хранятся в виде длинной спирали (см., например, рис. 2.21). В этом разделе будут рассматриваться диски DVD — более распространенные, чем Blu-ray, но диски Blu-ray очень похожи на DVD, не считая того, что для кодирования информации они используют формат MPEG-4 вместо MPEG-2. Во всех оптических накопителях головка воспроизведения должна следовать по спирали вращающегося диска. Цена таких устройств остается невысокой благодаря относительно простой механической конструкции и жесткому программному контролю положения головки. С головки поступает аналоговый сигнал, который перед обработкой необходимо преобразовать в цифровой формат. После оцифровки проводится интенсивная программная коррекция ошибок, которые возникают при прессовке дисков. Видеоданные сохраняются на носителе в формате MPEG-2, для декомпрессии которого требуются сложные вычисления типа преобразований Фурье. Аудиоданные сжимаются по психоакустической модели, которая не менее сложна в части декомпрессии. Наконец, аудио- и видеоданные должны быть приведены в форму, подходящую для вывода сигнала в телевизорах системы NTSC, PAL или SECAM — в зависимости от страны, в которой используется DVD-плеер. Естественно, программно решить все эти задачи в реальном времени на дешевом универсальном процессоре невозможно. Таким образом, нужен гетерогенный мультипроцессор с несколькими специализированными ядрами. Логическая схема DVD-плеера представлена на рис. 8.10.

Ядра, изображенные на рис. 8.10, отличаются по функциональной специализации; каждое из них спроектировано с расчетом на достижение максимального результата при минимально возможной цене. К примеру, сжатый видеосигнал для DVD хранится в формате **MPEG-2** (это аббревиатура разработавшей данный формат организации Motion Picture Experts Group — группа экспертов в области движущихся изображений). При сжатии каждый кадр разделяется на несколько блоков и в отношении каждого из них выполняются сложные преобразования.

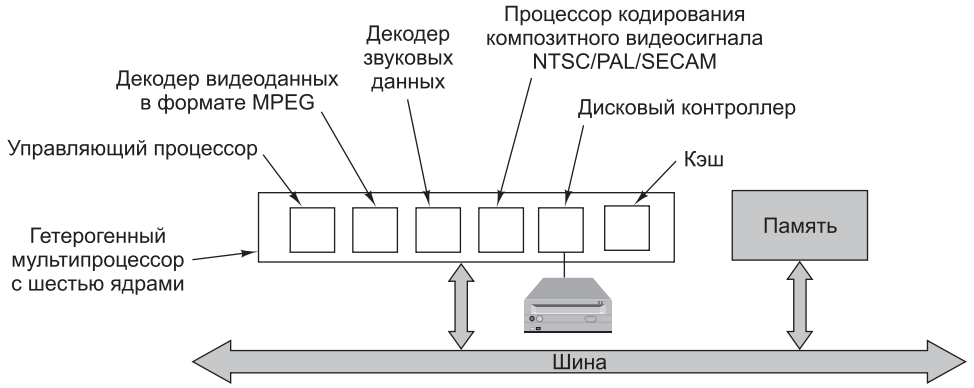


Рис. 8.10. Логическая схема простого DVD-плеера с гетерогенным мультипроцессором и несколькими специализированными ядрами для выполнения различных функций

Кадр может полностью состоять из измененных блоков или из блоков, присутствующих в предыдущем кадре с указанием смещения $(\Delta x, \Delta y)$ от текущего положения и измеренных пикселов. Программно подобные вычисления выполняются очень медленно, однако при наличии процессора декодирования MPEG-2 этот процесс значительно ускоряется. Аналогичным образом, декодирование и повторное кодирование композитного аудио-видеосигнала в соответствии с одним из стандартных телевизионных стандартов эффективнее проводить с помощью специализированного аппаратного процессора. В этом контексте совершенно естественно, что в DVD-плеерах и им подобных устройствах применяются гетерогенные мультипроцессоры с несколькими ядрами. В то же время, поскольку управляющий процессор представляет собой универсальное программируемое устройство, такую мультипроцессорную микросхему можно установить в близком в функциональном плане устройстве, например в устройстве записи DVD.

Гетерогенные мультипроцессоры также устанавливаются в моделях сотовых телефонов (CDMA или GSM), укомплектованных фото- и видеокамерами, игровыми приложениями, браузерами, клиентами электронной почты, приемниками цифрового спутникового сигнала и средствами беспроводного подключения к Интернету (IEEE 802.11, или WiFi). Сейчас не все телефоны оснащены этими функциями, но в будущем они, вероятно, распространятся повсеместно. По мере того как в современных условиях устройства постоянно усложняются, часы становятся GPS-навигаторами, а солнечные очки — радиоприемниками, потребность в гетерогенных мультипроцессорах только растет.

Довольно скоро обычным явлением станут микросхемы из десятков миллиардов миллионов транзисторов. Проектировать столь громоздкие устройства в цельном варианте очень сложно — к моменту завершения работ они имеют все шансы устареть. Значительно разумнее разместить несколько ядер (которые, фактически, представляют собой библиотеки) с относительно большим числом транзисторов на одной микросхеме и объединить их. При этом разработчики должны принять решение о том, какой процессор будет управляющим, а какие — специализированными. Увеличение нагрузки на программную часть управляющего процессора замедляет работу системы, но удешевляет и уменьшает размер микросхемы. Наличие нескольких специализированных процессоров для обработки звуковых

и видеоданных требует увеличения площади микросхемы и повышает ее стоимость, но, с другой стороны, обеспечивает высокую производительность при относительно низкой тактовой частоте, в связи с чем снижаются энергопотребление и теплоотдача. Иными словами, проектируя мультипроцессоры с несколькими ядрами, разработчики думают не о том, где разместить дополнительные транзисторы, а о том, каким образом и в каких вопросах можно идти на компромисс.

Программы обработки звуковых и видеоданных работают с огромными объемами информации. Поскольку все эти данные требуется обрабатывать быстро, от 50 до 75 % площади микросхемы отводится под размещение того или иного типа памяти. В этой связи возникают многочисленные вопросы. Сколько уровней кэш-памяти нужно в том или ином случае? Какими должны быть модули кэш-памяти, раздельными или объединенными? Каким должен быть объем каждого модуля? Насколько быстро он должны работать? Нужно ли размещать на микросхеме модули памяти других видов? Каких именно: SRAM или, может быть, SDRAM? От ответов на эти вопросы во многом зависит производительность, энергопотребление и параметры тепловыделения микросхемы.

Помимо процессоров и памяти, необходимо разработать схему взаимодействия ядер друг с другом. В небольших системах для этой цели вполне подойдет единственная шина, однако в более крупных системах такое решение может привести к тому, что схема взаимодействия ядер окажется узким местом всей системы. Во многих случаях проблема решается установкой нескольких шин или организацией кольцевой топологии. В последнем случае арбитраж осуществляется путем отправки по кольцу небольшого пакета — так называемого **маркера**, или **токена** (token). Перед передачей данных ядро должно удержать полученный токен. Завершив передачу, ядро пускает токен далее по кругу. Таким образом, исключаются конфликты при передаче данных.

В качестве примера механизма взаимодействия ядер на микросхеме рассмотрим архитектуру **CoreConnect** компании IBM (рис. 8.11). Она предназначена для объединения ядер в однокристальных гетерогенных мультипроцессорах. Для однокристальных мультипроцессоров CoreConnect исполняет примерно ту же роль, что и шина PCI для Pentium. С другой стороны, в отличие от PCI архитектура CoreConnect разрабатывалась без расчета на обратную совместимость с устаревшими аппаратными компонентами и протоколами и без оглядки на ограничения, связанные с передачей данных на уровне плат, в частности с количеством выводов на торцевых соединителях.

Архитектура CoreConnect состоит из трех шин. **Шина процессора** представляет собой высокоскоростную синхронную конвейеризированную шину с 32, 64 или 128 информационными линиями, работающими на тактовой частоте 66, 133 или 183 МГц. Ее максимальная пропускная способность равна 23,4 Гбит/с (для сравнения, у шины PCI этот показатель составляет 4,2 Гбит/с). Конвейеризация позволяет ядрам запрашивать шину в процессе передачи данных. Кроме того, как и в шине PCI, ядра могут одновременно передавать данные по разным линиям. Шина процессора оптимизирована для передачи коротких блоков данных и призвана обеспечивать взаимодействие между быстрыми ядрами — процессорами, декодерами MPEG-2, высокоскоростными сетями и тому подобными устройствами.

Поскольку одной шины процессора на всю микросхему недостаточно, для передачи данных между низкоскоростными устройствами ввода-вывода (UART,

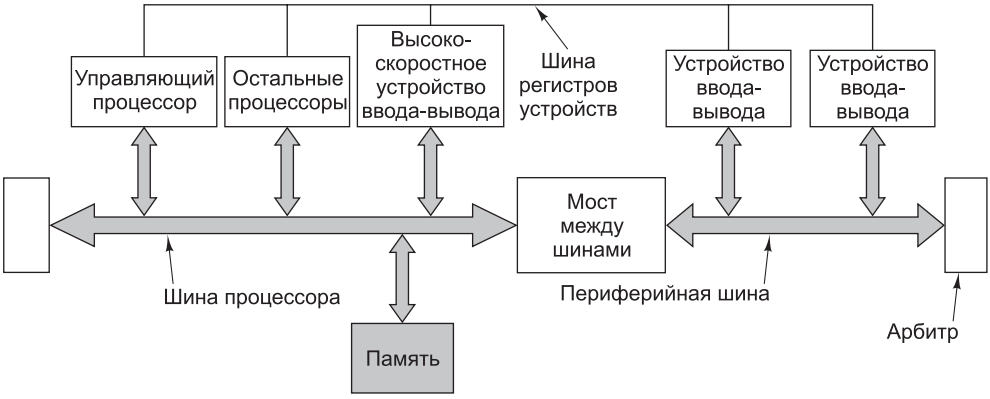


Рис. 8.11. Архитектура CoreConnect компании IBM

таймерами, USB-контроллерами, последовательными устройствами ввода-вывода и т. д.) предусмотрена вторая, **периферийная**, шина. Она упрощает взаимодействие между 8-, 16- и 32-разрядными периферийными устройствами, используя для этой цели всего несколько сотен вентилей. Периферийная шина также является синхронной, а ее максимальная пропускная способность достигает 300 Мбит/с. Эти две шины соединяются с помощью моста, напоминающего мосты, которыми до недавнего времени соединялись шины PCI и ISA, пока некоторое время назад шина ISA не была вытеснена окончательно.

В архитектуре CoreConnect есть также **шина регистров устройств**. Это крайне медленная асинхронная шина квитирования, позволяющая процессорам обращаться к регистрам периферийных устройств с целью управления этими устройствами. Передачи по ней проводятся нерегулярно, по несколько байтов.

Сочетание стандартной шины на микросхеме, интерфейса и подобающей инфраструктуры позволяет рассматривать CoreConnect как миниатюрную версию архитектуры PCI, для которой в перспективе можно наладить производство совместимых процессоров и контроллеров. Разница лишь в том, что в мире PCI производители разрабатывают и продают платы продавцам и конечным пользователям. В случае с CoreConnect разработчики ядер предоставляют лицензии на их производство изготовителям бытовой электроники и другим компаниям, которые затем разрабатывают гетерогенные мультипроцессоры на основе собственных и лицензированных ядер. Поскольку для производства больших и сложных микросхем требуются масштабные инвестиции в производственные мощности, в большинстве случае изготовители бытовой электроники готовят проекты в расчете на заказ микросхем в специализированных компаниях. В настоящее время существуют ядра для процессоров различных типов (ARM, MIPS, PowerPC и т. д.), декодеров MPEG, цифровых процессоров сигналов и всех стандартных контроллеров ввода-вывода.

Помимо CoreConnect, существуют и другие архитектуры встраиваемых в микросхему шин. IIIина **AMBA** (Advanced Microcontroller Bus Architecture также широко применяется для соединения процессоров ARM с другими процессорами и устройствами ввода-вывода [Flynn, 1997]. Несколько менее популярны шины **VCI** (Virtual Component Interconnect — взаимодействие виртуальных компонентов) и **OCP-IP** (Open Core Protocol-International Partnership — международный

консорциум по открытому протоколу ядра) — см. [Bhakthavatchalu et al., 2010]. Впрочем, встроенные в микросхему шины — это только начало; уже сейчас проектируются целые сети на одной микросхеме [Ahmadinia and Shahrabi, 2011].

Поскольку производителям микросхем не удается постоянно повышать тактовую частоту из-за проблем, связанных с тепловыделением, разработка однокристальных мультипроцессоров становится крайне актуальной. Дополнительные сведения по этой теме можно получить в [Gupta et al., 2010; Herrero et al., 2010; Mishra et al., 2011].

Сопроцессоры

Разобравшись с методами реализации внутрипроцессорного параллелизма, рассмотрим варианты повышения быстродействия компьютера за счет введения второго, специализированного процессора. Такие **сопроцессоры** очень разнообразны, в том числе по масштабу. В мэйнфреймах IBM 360 и во всех последующих моделях этой линейки для ввода-вывода предусмотрены независимые каналы. В составе CDC 6600 было 10 независимых процессоров, отвечавших за ввод-вывод. Другая область применения сопроцессоров — обработка графики и арифметические операции с плавающей точкой. Даже микросхему DMA можно рассматривать как сопроцессор. Иногда процессор передает сопроцессору на исполнение команду или набор команд; в иных случаях сопроцессор действует независимо и исполняет собственные команды.

Конструктивно сопроцессоры могут быть выполнены в отдельном корпусе (пример — каналы ввода-вывода IBM 360), в виде подключаемой платы (как в сетевых процессорах) или установлены на основной микросхеме (таковы сопроцессоры для обработки чисел с плавающей точкой). Как бы то ни было, все эти варианты объединяет подчиненная роль сопроцессора по отношению к основному процессору. Далее мы рассмотрим три области, в которых применение сопроцессоров приводит к ощутимому повышению быстродействия: сетевую поддержку, обработку мультимедийных данных и криптографию.

Сетевые процессоры

Большинство современных компьютеров подключены к локальной сети или Интернету. В результате технологического прогресса в области сетевого оборудования передача данных в сетях сегодня происходит так быстро, что программно обрабатывать все входящие и исходящие данные становится все сложнее. В связи с этим разрабатываются специальные сетевые процессоры, предназначенные для обработки трафика, и ими уже сегодня оснащаются многие высокопроизводительные вычислительные системы. В этом подразделе мы вкратце напомним основы организации сетей и обсудим принципы работы сетевых процессоров.

Основные сведения о сетях

Вычислительные сети подразделяются на два основных вида: **локальные сети** (Local Area Networks, **LAN**) соединяют компьютеры, расположенные в одном помещении или здании, а **глобальные сети** (Wide Area Networks, **WAN**) связывают

компьютеры, находящиеся на значительном расстоянии друг от друга. Наиболее популярный тип локальных сетей называется Ethernet. Первоначально простая сеть Ethernet состояла из толстого кабеля, к которому с помощью устройства под ироничным названием **зуб вампира** (vampire tap) подводились провода от входящих в сеть компьютеров. В современных сетях Ethernet компьютеры подключаются к центральному коммутатору (он показан справа на рис. 8.12). В первых версиях Ethernet скорость передачи данных ограничивалась значением 3 Мбит/с, а в первой коммерческой версии она увеличилось до 10 Мбит/с. Впоследствии появились версии Fast Ethernet и Gigabit Ethernet со скоростями передачи 100 Мбит/с и 1 Гбит/с соответственно. Не так давно выпущена коммерческая версия со скоростью 10 Гбит/с, а в недалеком будущем скорость возрастет до 40 Гбит/с.

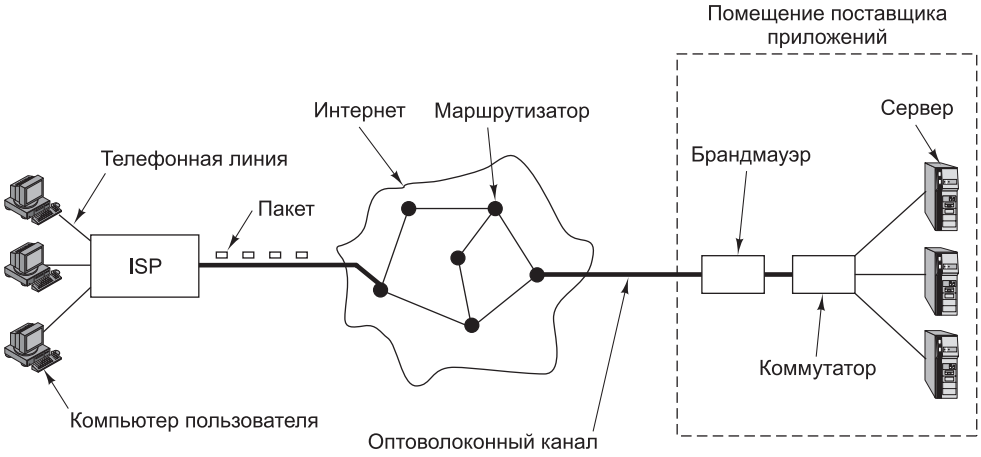


Рис. 8.12. Схема подключения пользователей к серверам Интернета

Организация глобальных сетей не столь единообразна. В таких сетях устанавливаются компьютеры, называемые **маршрутизаторами**, между которыми прокладываются проводные или оптоволоконные каналы. От исходного компьютера к целевому данные передаются в виде небольших блоков по 64–1500 байт, называемых **пакетами**. На каждом ретрансляционном участке, или переходе, пакет сначала сохраняется в памяти маршрутизатора, а затем отправляется дальше, как только освободится необходимая линия связи. Такой подход называется **коммутацией с сохранением и продвижением пакетов** (store-and-forward packet switching).

Хотя многим кажется, что Интернет является глобальной сетью, технически он представляет собой объединение большого числа разнообразных сетей. Впрочем, это различие для дальнейшего рассмотрения непринципиально. На рис. 8.12 структура Интернета показана с точки зрения домашнего пользователя. Компьютер такого пользователя обычно обращается к веб-серверам через телефонную линию, подключаясь к ней либо при помощи обычного модема, либо через линию ADSL (см. главу 2). Для подключения может также использоваться кабель кабельного телевидения. В этом случае правая часть рис. 8.12 будет выглядеть немного иначе, поскольку поставщиком приложений будет компания кабельного телевидения. Итак, компьютер пользователя разбивает данные на па-

кеты, чтобы передать их специальной компании, которая называется **поставщиком услуг Интернета** (Internet Service Provider, **ISP**). Сам поставщик услуг Интернета использует высокоскоростное соединение (обычно по оптоволокну) с одной из региональных или магистральных сетей, составляющих Интернет. Пакеты передаются между компьютерами сети, перемещаясь переход за переходом к серверу.

В большинстве компаний, предлагающих веб-услуги, имеется так называемый **брандмауэр** — специализированный компьютер, фильтрующий входящие пакеты, удаляя нежелательные (например, пакеты от хакеров). Брандмауэр подключается к местной локальной сети (обычно через коммутатор), которая отвечает за доставку пакетов требуемому серверу. Естественно, реальность значительно сложнее, но основные идеи, которые иллюстрирует рис. 8.12, верны.

Сетевое программное обеспечение состоит из нескольких **протоколов**, каждый из которых представляет собой набор форматов, последовательностей обмена и правил, определяющих назначение пакетов. Например, когда пользователь хочет получить веб-страницу с сервера, его браузер по протоколу **HTTP** (HyperText Transfer Protocol — **протокол передачи гипертекста**) отправляет пакет с запросом `GET PAGE` серверу, который «знает», как обрабатывать полученный пакет. В процессе передачи используются множество различных протоколов, причем зачастую совместно. Обычно они организованы в иерархическую многоуровневую структуру, в которой протоколы верхнего уровня передают пакеты протоколам нижележащих уровней, а реальной передачей данных занимается протокол самого нижнего уровня. На стороне получателя пакеты поднимаются вверх по иерархии в обратном порядке.

Раз сетевые процессоры заняты выполнением протоколов, перед изучением процессоров имеет смысл поговорить о протоколах немного подробнее. Вернемся ненадолго к запросу `GET PAGE`. Как именно он попадает на веб-сервер? Прежде всего браузер устанавливает с сервером соединение по протоколу **TCP** (Transmission Control Protocol — **протокол управления передачей**). Программное обеспечение, реализующее этот протокол, следит за тем, чтобы все отправленные пакеты были доставлены, причем в правильном порядке. В случае потери пакета программное обеспечение TCP максимально быстро повторяет передачу до тех пор, пока пакет наконец не будет получен.

Реально происходит следующее. Браузер формирует корректное HTTP-сообщение с запросом `GET PAGE`, а затем передает его программному обеспечению TCP, которое и передает пакет через соединение. Программным обеспечением TCP в начало сообщения добавляется заголовок, содержащий порядковый номер и другую информацию. Этот дополнительный заголовок называется **TCP-заголовком**.

Сделав свою часть работы, программное обеспечение TCP передает TCP-заголовок вместе с полезной нагрузкой (содержащей запрос `GET PAGE`) еще одной программе, реализующей протокол **IP** (Internet Protocol — **межсетевой протокол**). Эта программа добавляет в начало пакета IP-заголовок с информацией об адресах отправителя (то есть машины, передающей пакет) и получателя (машины, ожидающей пакет), максимальном числе переходов, двигаясь вдоль которых пакет будет существовать (чтобы «потерявшиеся» пакеты не жили вечно, заполоняя собой весь Интернет), контрольной суммой (для обнаружения ошибок памяти и ошибок передачи) и рядом других полей.

Далее пакет (включающий в себя IP-заголовок, TCP-заголовок и сам запрос `GET PAGE`) передается «вниз» на уровень канала передачи данных, который

добавляет к пакету свой заголовок и передает пакет по линии связи. Этот уровень также дописывает в конец контрольную сумму, называемую **CRC** (Cyclic Redundancy Check — **циклический контроль избыточности**) и позволяющую выявлять ошибки передачи. Может показаться, что две контрольные суммы, на уровне IP и на уровне канала данных, — это больше, чем необходимо, но такой подход повышает надежность. На каждом переходе проверяется CRC-код пакета, после чего заголовок вместе с CRC-кодом генерируются заново, в соответствии с требованиями исходящего канала передачи данных. На рис. 8.13 показано, что представляет собой пакет в Ethernet. Случай телефонной линии (ADSL) отличается только тем, что вместо Ethernet-заголовка имеется заголовок коммутируемой линии. Обработка заголовков — одна из важных задач, которую призваны решать сетевые процессоры. Конечно, излишне упоминать о том, что мы лишь вскользь коснулись темы вычислительных сетей. За более детальными сведениями обращайтесь к [Tanenbaum and Wetherall, 2011].

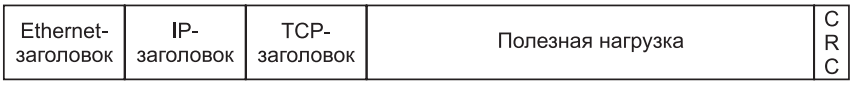


Рис. 8.13. Вид Ethernet-пакета

Основные сведения о сетевых процессорах

К сети подключаются самые разные устройства. Для конечных пользователей это, прежде всего, персональные компьютеры (настольные и ноутбуки), но растет также число игровых консолей, персональных электронных секретарей (карманных компьютеров), сотовых телефонов. Для компаний роль оконечных систем играют серверы и персональные компьютеры. Помимо этого, в сетях функционирует бесчисленное количество разнообразных промежуточных устройств, в их число входят маршрутизаторы, коммутаторы, брандмауэры, прокси-серверы, системы балансировки нагрузки. Достаточно любопытно, что к этим промежуточным системам предъявляются самые серьезные требования — именно они должны обеспечивать передачу максимального числа пакетов в секунду. Кроме того, серьезные требования налагаются на серверы, что касается пользовательских компьютеров, то к ним особых требований нет.

В зависимости от сети и самого пакета, поступающий в сеть пакет перед отправкой по исходящей линии или передачей прикладной программе может требовать той или иной обработки. Обработка может включать принятие решения о том, куда передавать пакет, разбиение пакета на части или сборку его из частей, управление качеством обслуживания (особенно в отношении аудио- и видеопотоков), защиту данных (кодировании и декодирование), компрессию и декомпрессию и т. п.

Когда скорость передачи данных в локальной сети приближается к 40 Гбит/с, а размер пакета — к 1 Кбайт, сетевой компьютер должен обрабатывать почти 5 млн пакетов в секунду. Для пакетов размером 64 байт это значение возрастает примерно до 80 млн пакетов в секунду. Выполнение всех упомянутых функций за время порядка 12–200 нс, в дополнение к неизменно необходимому копированию пакетов, просто невозможно реализовать программно. Аппаратная поддержка здесь принципиально необходима.

Одним из путей аппаратного решения проблемы быстрой обработки пакетов является использование **специализированных интегральных схем** (Application-

Specific Integrated Circuit, **ASIC**). Такая микросхема подобна аппаратно реализованной программе, которая может выполнять любое из заранее предусмотренных действий. Основой многих современных маршрутизаторов являются схемы ASIC. Впрочем, и со специализированными интегральными схемами связаны некоторые проблемы. Прежде всего, их долго проектировать и не менее долго производить. Кроме того, это жестко запрограммированные устройства, то есть чтобы внести новую функциональность, приходится разрабатывать и изготавливать новую микросхему. Хуже того, настоящим кошмаром являются ошибки, так как единственным способом их исправления является разработка, изготовление и установка новой (исправленной) микросхемы. Наконец, этот подход является весьма затратным, если только большой объем производства не позволяет компенсировать расходы на разработку.

Второй подход основан на использовании **программируемых вентильных матриц** (Field Programmable Gate Array, **FPGA**). Такая матрица представляет собой набор вентилей, из которых путем перекоммутации строится требуемая схема. Сроки выхода программируемых вентильных матриц на рынок гораздо короче, чем у специализированных интегральных схем, к тому же их можно перепрограммировать в «полевых условиях» при помощи специального программатора. Но в то же время они очень сложные, дорогие и более медленные, чем схемы ASIC, поэтому программируемые вентильные матрицы не получили широкого распространения, исключая некоторые узкоспециальные области.

Наконец, перейдем к **сетевым процессорам** — устройствам, способным обрабатывать входящие и исходящие пакеты со скоростью их передачи, то есть в реальном времени. Обычно они реализуются в виде съемной платы, содержащей, помимо кристалла сетевого процессора, память и вспомогательную логику. К плате подключается одна или несколько сетевых линий. Процессор получает из линии пакеты, обрабатывает их, после чего передает по другой линии, если это маршрутизатор, или отправляет в главную системную шину (то есть в шину PCI), если это оконечное устройство, которым может быть, например, персональный компьютер. Типичный сетевой процессор и его плата показаны на рис. 8.14.

Обычно на плате имеется как статическая (SRAM), так и синхронная динамическая оперативная память (SDRAM) — эти виды памяти применяются для разных целей. SRAM быстрее SDRAM, но из-за дороговизны памяти этого типа обычно немного. Она используется для хранения таблиц маршрутизации и прочих ключевых структур данных, в то время как в SDRAM записываются сами обрабатываемые пакеты. Благодаря тому, что память обоих этих типов располагается вне кристалла сетевого процессора, можно гибко подойти к вопросу о выборе объема памяти. Так, в простых системах с единственной сетевой линией (такие платы могут ставиться, например, в персональный компьютер или сервер) памяти может быть немного, в то время как маршрутизатору ее требуется намного больше.

Сетевые процессоры оптимизированы для быстрой обработки большого количества входящих и исходящих пакетов. Это означает, что по каждой из сетевых линий проходят миллионы пакетов в секунду, а маршрутизатор должен поддерживать десятки таких линий. Столь серьезных показателей можно достигнуть, только на процессорах с высокой степенью внутреннего параллелизма. Кроме того, в процессор обязательно входят несколько **PPE-контроллеров** (Protocol/Programmable/Packet Processing Engine — программируемая система обработки

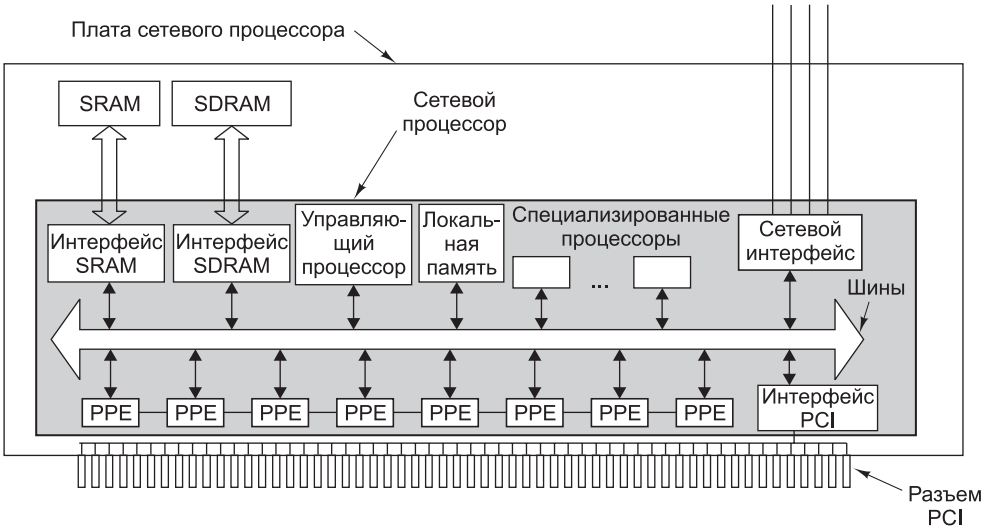


Рис. 8.14. Кристалл и плата типичного сетевого процессора

пакетов и протоколов), каждая из которых состоит из RISC-ядра (возможно, модифицированного) и внутренней памяти небольшого объема для хранения программы и нескольких переменных.

Есть два подхода к организации PPE-контроллеров. В простейшем случае все PPE-контроллеры делаются идентичными. Когда в сетевой процессор приходит новый пакет, он передается для обработки тому PPE-контроллеру, который в данный момент бездействует. Если свободных PPE-контроллеров нет, пакет ставится в очередь в расположенной на плате памяти SDRAM, ожидая освобождения одного из PPE-контроллеров. При такой организации горизонтальные связи, показанные на рис. 8.14, отсутствуют, так как у разных PPE-контроллеров нет необходимости общаться друг с другом.

Другой подход к организации PPE-контроллеров — конвейер, где каждый из PPE-контроллеров выполняет один этап обработки, после чего передает указатель на полученный пакет следующему PPE-контроллеру в конвейере. Такой конвейер работает во многом аналогично конвейерам ЦП, которые рассматривались в главе 2. В обоих вариантах организации PPE-контроллеры являются полностью программируемыми.

В более совершенных сетевых процессорах PPE-контроллеры поддерживают многопоточность, то есть каждый из них имеет несколько наборов регистров и аппаратный регистр, показывающий, какой из наборов используется. Это позволяет одновременно выполнять несколько программ (то есть программных потоков) и переключаться между ними, просто изменяя переменную «текущего рабочего набора регистров». Когда один из программных потоков вынужден ждать (например, при обращении к SDRAM, на которое требуется несколько циклов), PPE-контроллер может быть мгновенно переключен на поток, способный продолжать работу. Это позволяет добиваться высокой загрузки PPE-контроллеров, даже несмотря на необходимость часто ожидать завершения обмена данными с SDRAM или других медленных внешних операций.

Помимо PPE-контроллеров, у всех сетевых процессоров имеется управляющий процессор для выполнения всех действий, не относящихся напрямую к обработке пакетов (например, обновление таблиц маршрутизации). Обычно он представляет собой RISC-процессор общего назначения, память для данных и команд которого находится на одном кристалле с процессором. Более того, в сетевом процессоре может быть несколько специализированных процессоров, предназначенных для выполнения критически важных операций. Они представляют собой очень маленькие специализированные интегральные схемы (ASIC), способные выполнять только одно несложное действие, такое как поиск целевого адреса в таблице маршрутизации. Все компоненты сетевого процессора взаимодействуют друг с другом на мультигигабитных скоростях по одной или нескольким расположенным на кристалле параллельным шинам.

Обработка пакетов

Независимо от того, имеет сетевой процессор конвейерную или параллельную организацию, каждый прибывший пакет проходит несколько этапов обработки. У некоторых процессоров эти этапы разделяются на **входную** (ingress processing) и **выходную обработку** (egress processing). К первой группе относятся операции с пакетами, пришедшими извне (через сетевую линию или системную шину), а ко второй — с пакетами перед их отправкой. Таким образом, каждый пакет подвергается сначала входной обработке, а затем выходной. Это разделение достаточно условно, так как некоторые операции можно производить на любом из этапов (например, сбор сведений о трафике).

Мы рассмотрим эти этапы в том порядке, в котором они могли бы выполняться, но имейте в виду, что выполнять их нужно не для всех пакетов, кроме того, возможна другая последовательность действий.

1. *Проверка контрольной суммы.* Если входящий пакет прибывает из сети Ethernet, его контрольная сумма (CRC-код) пересчитывается и сравнивается со значением, имеющемся в пакете, чтобы убедиться, что пакет принят без ошибок. Если оба значения оказываются равными или поле CRC в Ethernet-пакете отсутствует, вычисляется контрольная сумма IP-пакета и сравнивается со значением в пакете. Это позволяет убедиться, что IP-пакет не был поврежден по вине сбойного бита в памяти отправителя после того, как отправителем была посчитана контрольная сумма для IP-пакета. Если все проверки пройдены, пакет передается для дальнейшей обработки, в противном случае он просто отбрасывается.
2. *Извлечение значений полей.* Путем анализа определяется положение нужного заголовка, и из пакета извлекаются значения соответствующих этому заголовку ключевых полей. В Ethernet-коммутаторе исследуется только Ethernet-заголовок, в IP-маршрутизаторе — только IP-заголовок. Значения ключевых полей сохраняются либо в регистрах (при параллельной организации PPE-контроллеров), либо в SRAM (при конвейерной организации).
3. *Классификация пакетов.* Пакеты классифицируются в соответствии с рядом программных правил. В простейшем случае пакеты данных отделяются от управляющих пакетов, но обычно разделение гораздо более тонкое.
4. *Выбор пути.* У большинства сетевых процессоров имеется особый быстрый путь, оптимизированный для передачи всего многообразия пакетов данных, в то время как остальные пакеты обрабатываются по-своему, обычно управ-

ляющим процессором. Соответственно выбирается либо быстрый путь, либо один из медленных путей.

5. *Определение целевой сети.* В IP-пакетах содержится 32-разрядный адрес получателя. Однако невозможно (и нежелательно) использовать для поиска получателя каждого пакета целую таблицу из 232 записей. Поэтому левая часть адреса обычно содержит адрес сети, а правая указывает на отдельную машину в этой сети. Длина адреса сети не фиксирована, поэтому его определение — задача нетривиальная, причем ее еще больше усложняет тот факт, что допустимо несколько вариантов, из которых правильным считается самый длинный. На этом шаге часто применяется специализированная интегральная схема.
6. *Поиск маршрута.* После определения адреса целевой сети по таблице, хранящейся в памяти SRAM, выясняется, по какой из исходящих линий отправить пакет. Опять же, на этом шаге может применяться специализированная интегральная схема.
7. *Разбивка и сборка.* Приложения часто максимально увеличивают полезную нагрузку (данные) TCP-пакетов, стараясь сократить количество системных вызовов, но и у TCP, и у IP, и у Ethernet есть ограничения на максимальный размер пакета. Как следствие этих ограничений, может потребоваться разбивать пакеты (и, соответственно, полезную нагрузку) на части перед отправкой и собирать их снова на приемной стороне. Эти функции может выполнять сетевой процессор.
8. *Вычисления.* Иногда требуется производить над данными те или иные сложные вычисления, например выполнять компрессию и декомпрессию, кодирование и декодирование. Эти действия можно переложить на сетевой процессор.
9. *Управление заголовками.* Иногда приходится добавлять или удалять заголовки, а также менять значения тех или иных полей. Например, в IP-заголовке есть счетчик числа хопов, которые пакет еще может пройти перед самоуничтожением. После прохождения каждого перехода значение счетчика необходимо уменьшать на единицу, и эту функцию вполне может выполнять сетевой процессор.
10. *Управление очередями.* Входящим и исходящим пакетам часто приходится становиться в очередь в ожидании обработки. Но для мультимедийных приложений во избежание джиттера необходимо, чтобы задержки между пакетами не превышали определенного значения. Кроме того, брандмауэру или маршрутизатору может потребоваться перераспределять входную нагрузку между несколькими выходными линиями по определенным правилам. Все эти задачи могут решаться сетевым процессором.
11. *Генерирование контрольных сумм.* В исходящих пакетах должны быть контрольные суммы. Контрольная сумма IP-пакетов может вычисляться сетевым процессором, в то время как контрольная сумма Ethernet-пакетов в общем случае генерируется аппаратно.
12. *Учет.* В некоторых случаях необходимо подсчитывать трафик при прохождении пакетов, особенно если одна из сетей в качестве коммерческой услуги предлагает транзит трафика. Учетом может заниматься сетевой процессор.
13. *Сбор статистики.* Многие компании хотели бы иметь статистику трафика: сколько пакетов поступило, сколько пакетов ушло, когда это происходило и т. д. Сетевые процессоры могут собирать эти сведения.

Повышение производительности

Производительность — самая главная характеристика сетевых процессоров. Что можно сделать для ее повышения? Прежде чем ответить на этот вопрос, необходимо определить, что это такое. Одной из метрик является количество пакетов, передаваемых в секунду, другой — количество байтов, предаваемых в секунду. Эти метрики предполагают разные подходы, и схема, хорошо работающая с маленькими пакетами, может плохо справляться с большими. В частности, при передаче маленьких пакетов заметно повысить производительность можно путем ускорения процесса поиска целевого адреса в таблице, в то же время при передаче больших пакетов заметного повышения производительности это не даст.

Самым прямым путем к повышению производительности является увеличение тактовой частоты сетевого процессора. Правда, производительность не растет пропорционально частоте, так как сказывается время обращения к памяти и ряд других факторов. Кроме того, большая частота означает необходимость отводить больше тепла.

Обычно выходом является увеличение числа PPE-контроллеров — этот подход особенно эффективен для параллельных архитектур. Может помочь также увеличение длины конвейера, но только в том случае, если удастся разделить процесс обработки пакета на достаточно простые этапы.

Еще один подход состоит в увеличении числа дополнительных специализированных процессоров или специализированных интегральных схем, предназначенных для выполнения отдельных затратных и часто требующихся операций, если такие операции эффективнее выполнять аппаратно. Среди множества кандидатов можно отметить поиск в таблицах, вычисление контрольных сумм и криптографические операции.

Можно также повысить скорость, сократив время прохождения пакетов в системе за счет введения дополнительных шин и увеличения количества линий в существующих. Наконец, обычно прироста производительности удается добиться заменой микросхем памяти (SRAM вместо SDRAM), но это, естественно, сказывается на стоимости.

Конечно же, это далеко не все, что можно сказать о сетевых процессорах. За дополнительной информацией обращайтесь к [Freitas et al., 2009; Lin et al., 2010; Yamamoto and Nakao, 2011].

Графические процессоры

Вторая область применения сопроцессоров — обработка графики высокого разрешения (например, генерирование 3D-изображений). Обычные процессоры недостаточно хорошо эффективны при сложных вычислениях с большими объемами данных. По этой причине некоторые современные персональные компьютеры и большинство разрабатываемых моделей оборудуются специальными сопроцессорами для обработки графики, на которые можно переложить значительную часть работы.

Графический процессор NVIDIA Fermi

Область обработки графики, значимость которой непрерывно возрастает, мы изучим на примере NVIDIA Fermi — архитектуры, применяемой в семействе графических процессоров с разными скоростями и размерами. Архитектура

графического процессора Fermi изображена на рис. 8.15. Она состоит из 16 **потоковых мультипроцессоров**, или SM (Streaming Multiprocessors), каждый из которых имеет собственный кэш первого уровня с высокой пропускной способностью. Каждый потоковый мультипроцессор содержит 32 ядра CUDA; таким образом, графический процессор Fermi в совокупности содержит 512 ядер CUDA. Ядро CUDA (Compute Unified Device Architecture) представляет собой простой процессор с поддержкой целочисленных вычислений с одинарной точностью и вычислений с плавающей точкой. Один блок SM с 32 ядрами CUDA изображен на рис. 2.6. 16 блоков SM совместно используют один объединенный 768-килобайтный кэш второго уровня, соединенный с многопортовым интерфейсом DRAM. Интерфейс управляющего процессора обеспечивает канал связи между управляющей системой и графическим процессором по общей шине DRAM (обычно через интерфейс PCI-Express).

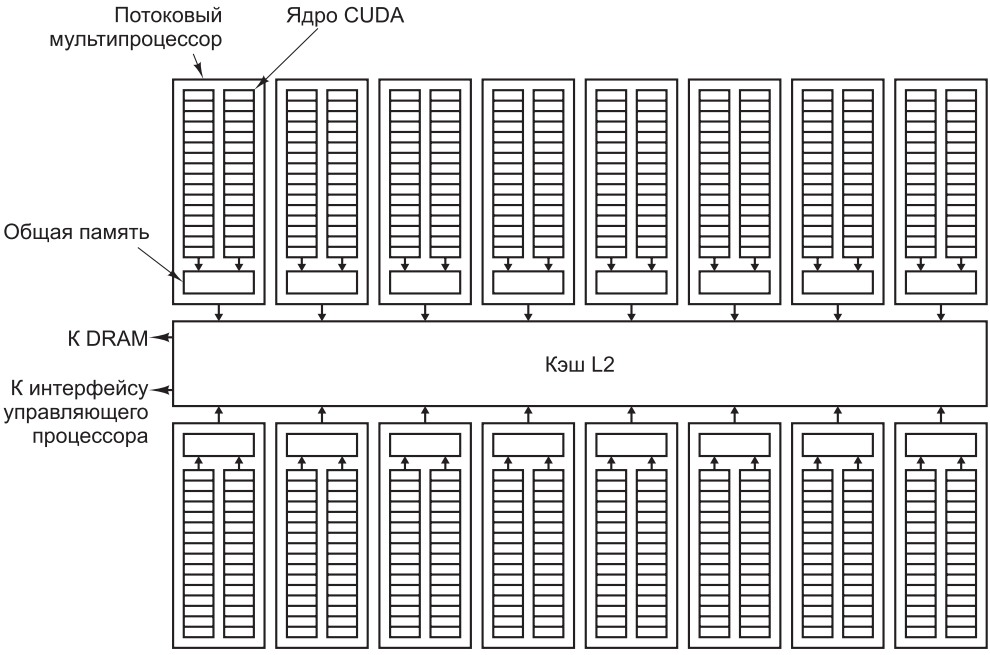


Рис. 8.15. Архитектура графического процессора Fermi

Архитектура Fermi спроектирована для эффективного выполнения кода обработки графических и видеоданных, в которых обычно задействованы значительные избыточные вычисления, распределенные между многими пикселами. Вследствие этой избыточности потоковые мультипроцессоры, способные выполнять до 16 операций за раз, требуют, чтобы все операции, выполняемые за один цикл, были идентичны. Такой стиль обработки называется **SIMD** (Single-Instruction Multiple Data); его важное преимущество заключается в том, что каждый блок SM осуществляет выборку и декодирует всего одну команду за цикл. Только за счет совместной обработки команд всеми ядрами SM компании NVIDIA удалось разместить 512 ядер на одной кремниевой подложке. Если программисты смогут в полной мере использовать всю вычислительную мощь Fermi (хотя это

«если» всегда выглядит неопределенно), система будет существенно превосходить такие традиционные скалярные архитектуры, как Core i7 или OMAP4430.

Специфические требования SIMD в SM накладывают ограничения на код, который может выполняться программистами в этих блоках. Для одновременного выполенния всех 16 операций каждое ядро CUDA должно выполнять один и тот же код. Чтобы упростить задачу программиста, компания NVIDIA разработала язык программирования CUDA, в котором программный параллелизм определяется с использованием программных потоков.

Программные потоки объединяются в блоки, которые назначаются потоковым процессорам. Если каждый программный поток в блоке выполняет одну и ту же последовательность команд (то есть во всех ветвях принимаются одинаковые решения), одновременно могут выполняться до 16 операций (при условии, что до 16 потоков готовы к выполнению). Когда потоки разных SM принимают разные решения, возникает эффект снижения быстродействия, из-за которого программные потоки с разными путями выполняются последовательно. Этот эффект уменьшает степень параллелизма и замедляет работу графического процессора. К счастью, в области обработки графики существует широкий спектр операций, которые помогают избежать эффекта снижения быстродействия и достигнуть хорошей производительности. От выполнения в SIMD-архитектурах выигрывают многие разновидности кода: медицинская визуализация, финансовые прогнозы, анализ графов и т. д. С расширением спектра потенциальных применений графических процессоров за ними закрепилось новое название **GPGPU** (General-Purpose Graphics Processing Units).

Даже с 512 ядрами CUDA графический процессор Fermi будет парализован без значительной пропускной способности памяти. Для этого графический процессор Fermi реализует современную иерархию памяти, изображенную на рис. 8.15. Каждый блок SM использует как общую память, так и собственный кэш данных уровня 1. Общая память напрямую адресуется ядрами CUDA и обеспечивает быстрое совместное использование данных между потоками одного блока SM. Кэш уровня 1 ускоряет работу с данными в DRAM. В соответствии с широким спектром использования программных данных, эти блоки SM могут оснащаться либо 16-килобайтной общей памятью с 48-килобайтным кэшем уровня 1, либо 48-килобайтной общей памятью с 16-килобайтным кэшем уровня 1. Все блоки SM используют один объединенный кэш уровня 2 на 768 Кбайт. Кэш второго уровня предоставляет быстрый доступ к данным DRAM, не поместившимся в кэш уровня 1. Кэш второго уровня также обеспечивает совместный доступ к данным для разных SM, хотя этот режим работает намного медленнее совместного доступа в общей памяти SM. За кэшем второго уровня находится память DRAM, в которой хранятся остальные данные, графическая информация и текстуры, которые используются программами, выполняемыми на графическом процессоре Fermi. Эффективные программы должны по возможности избегать использования DRAM, так как на реализацию всего одного обращения могут уйти сотни циклов.

Для изобретательного программиста графический процессор Fermi может стать одной из самых мощных вычислительных платформ. Всего один графический процессор GTX 580 на базе Fermi, работающий на частоте 772 МГц с 512 ядрами CUDA, способен обеспечить производительность до 1,5 терафлопа при

потреблении 250 Вт мощности. Статистика становится еще более впечатляющей, если учесть, что средняя цена GTX 580 составляет менее 600 долларов. Для сравнения: в 1990-е годы самый быстрый компьютер в мире Cray-2 имел производительность 0,002 терафлопа при цене в 30 млн долларов. При этом он занимал комнату средних размеров и требовал системы жидкостного охлаждения для отведения 150 кВт потребляемой мощности. GTX 580 в 750 раз превосходит его по производительности, а стоит в 50 000 раз меньше и потребляет в 600 раз меньше энергии. Согласитесь, неплохо?

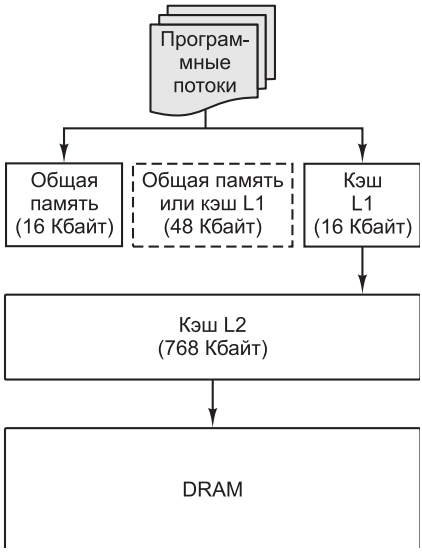


Рис. 8.16. Иерархия памяти графического процессора Fermi

Криптопроцессоры

Безопасность, а особенно сетевая безопасность — еще одна (уже третья) область, в которой широко используются сопроцессоры. Когда между клиентом и сервером устанавливается соединение, обычно требуется их взаимная аутентификация. По установленному таким образом безопасному (шифруемому) соединению можно безопасно передавать данные и не думать о злоумышленниках, прослушивающих линию.

Проблема здесь в том, что безопасность обеспечивается средствами криптографии, а эта область требует весьма объемных вычислений. В криптографии сейчас распространены два основных подхода к защите данных: **шифрование с симметричным ключом** и **шифрование с открытым ключом**. Первый основан на очень тщательном перемешивании битов (как будто сообщение помещают в некий электронный миксер). В основе второго подхода лежит умножение и возведение в степень больших чисел (1024-разрядных), что требует исключительно больших временны́х затрат.

Многими компаниями выпущены криптографические сопроцессоры, позволяющие шифровать данные для их безопасной передачи и потом расшифровывать их. Зачастую они представляют собой карты расширения, вставляемые

в PCI-разъем. Благодаря специальному аппаратному обеспечению, эти процессоры могут выполнять необходимые криптографические вычисления намного быстрее, чем центральный процессор. К сожалению, более детальное изучение криптографических процессоров потребовало бы уделить немало времени самой криптографии, что выходит за рамки темы этой книги. Дополнительную информацию о криптопроцессорах можно найти в [Gaspar et al., 2010; Haghighizadeh et al., 2010; Shoufan et al., 2011].

Мультипроцессоры

Мы выяснили, как ввести в однопроцессорную систему параллелизм, добавив в нее сопроцессор. Следующий шаг — объединение нескольких полноценных процессоров в одну большую систему. Такие системы с несколькими центральными процессорами можно разделить на мультипроцессоры и мультикомпьютеры. Сначала разобравшись со значением этих терминов, мы изучим мультипроцессоры, а вслед за ними — мультикомпьютеры.

Мультипроцессоры и мультикомпьютеры

В любой параллельной компьютерной системе процессоры, выполняющие разные части единого задания, должны как-то взаимодействовать друг с другом, чтобы обмениваться информацией. Как именно должен происходить обмен? Для этого было предложено и реализовано две стратегии: мультипроцессоры и мультикомпьютеры. Ключевое различие между стратегиями состоит в наличии или отсутствии общей памяти. Это различие сказывается как на конструкции, устройстве и программировании таких систем, так и на их стоимости и размерах.

Мультипроцессоры

Параллельный компьютер, в котором все процессоры совместно используют общую физическую память, называется **мультипроцессором**, или **системой с общей памятью** (рис. 8.17, *а*). Все процессы, работающие в мультипроцессоре совместно, могут иметь единое виртуальное адресное пространство, отображенное на общую память. Любой процесс с помощью команд `LOAD` и `STORE` может считать слово из памяти или записать слово в память. Больше ничего не требуется. Два процесса имеют возможность легко обмениваться информацией — для этого один из них просто записывает данные в общую память, а другой их считывает.

Благодаря возможности взаимодействия двух и более процессов мультипроцессоры весьма популярны. Данная модель понятна программистам и позволяет решать широкий круг задач. Для примера рассмотрим программу, которая анализирует битовое отображение и составляет список всех его объектов. Одна копия изображения хранится в памяти, как показано на рис. 8.17, *б*. Каждый из 16 процессоров запускает один процесс, призванный анализировать одну из 16 секций. Если процесс обнаруживает, что один из его объектов переходит через границу секции, этот процесс просто переходит вслед за объектом в следующую секцию, считывая слова этой секции. В нашем примере некоторые объекты обрабатываются несколькими процессами, поэтому в конце потребуется некоторая координация, чтобы определить количество домов, деревьев и самолетов.

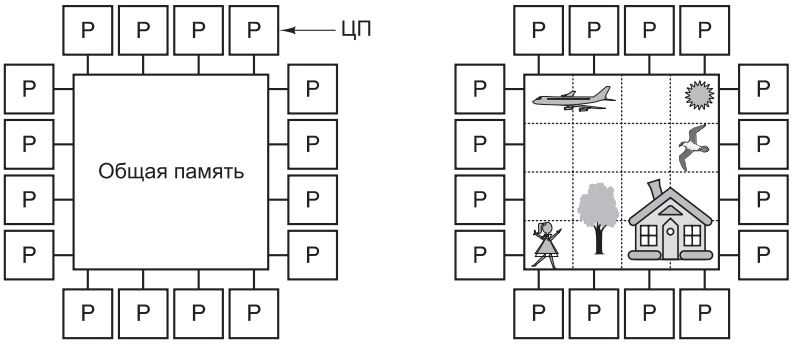


Рис. 8.17. Мультипроцессор из 16 процессоров, имеющих общую память (*а*); изображение, разделенное на 16 секций, каждую из которых анализирует отдельный процессор (*б*)

Поскольку все процессоры в мультипроцессоре используют единое адресное пространство, функционирует только одна копия операционной системы. Соответственно, имеется только одна карта страниц памяти и одна таблица процессов. Когда процесс блокируется, его процессор сохраняет свое состояние в таблицах операционной системы, а затем просматривает эти таблицы в поисках другого процесса, который нужно запустить. Именно такая организация, в основе которой лежит единая система, и отличает мультипроцессор от мультикомпьютера, в котором каждый компьютер имеет собственную копию операционной системы.

Мультипроцессор, как и все компьютеры, должен содержать устройства ввода-вывода (диски, сетевые адаптеры и т. п.). В одних мультипроцессорных системах только определенные процессоры получают доступ к устройствам ввода-вывода и, следовательно, обладают специальными средствами ввода-вывода. В других мультипроцессорных системах каждый процессор может получить доступ к любому устройству ввода-вывода. Если все процессоры имеют равный доступ ко всем модулям памяти и всем устройствам ввода-вывода и между процессорами возможна полная взаимозаменяемость, такой мультипроцессор называется **симметричным** (Symmetric MultiProcessor, **SMP**).

Мультикомпьютеры

Во втором варианте параллельной архитектуры каждый процессор имеет собственную память, доступную только этому процессору. Такая схема называется **мультикомпьютером**, или **системой с распределенной памятью** (рис. 8.18, *а*). Ключевое отличие мультикомпьютера от мультипроцессора состоит в том, что каждый процессор в мультикомпьютере имеет собственную локальную память, к которой этот процессор может обращаться, выполняя команды `LOAD` и `STORE`, но никакой другой процессор с помощью этих команд не может получить доступ к локальной памяти данного процессора. Таким образом, мультипроцессоры имеют одно физическое адресное пространство, разделяемое всеми процессорами, а мультикомпьютеры содержат отдельные физические адресные пространства для каждого процессора.

Поскольку процессоры в мультикомпьютере не могут взаимодействовать друг с другом простыми обращениями к общей памяти, процессоры обмениваются со-

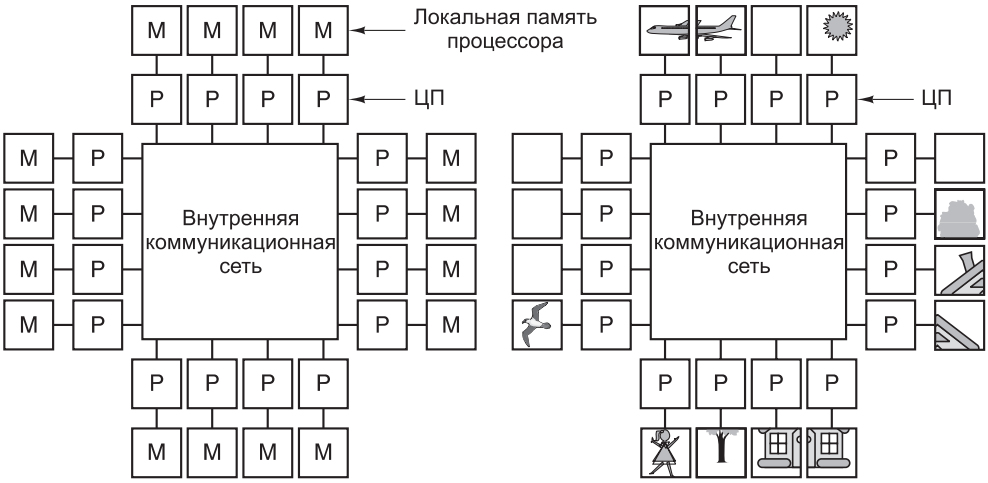


Рис. 8.18. Мультикомпьютер из 16 процессоров, каждый из которых имеет собственную память (*а*); битовая карта изображения с рис. 8.16, разделенная между 16 модулями памяти (*б*)

общениями через связывающую их коммуникационную сеть. В качестве примеров мультикомпьютеров можно назвать IBM BlueGene/L, Red Storm и кластер Google.

При отсутствии общей памяти, реализованной аппаратно, предполагается определенная программная структура. В мультикомпьютере невозможно иметь единое для всех процессоров виртуальное адресное пространство, позволяющее считывать и записывать информацию командами `LOAD` и `STORE`. Например, если процессор в верхнем левом углу рис. 8.17, *б* (присвоим этому процессору номер 0) обнаружит, что часть его объекта попадает в другую секцию, относящуюся к следующему процессору (пусть это будет процессор 1), он может просто продолжать считывать информацию из памяти, чтобы получить изображение хвоста самолета. Однако если то же самое обнаружит процессор 0 на рис. 8.18, *б*, он не сможет просто считать информацию из памяти процессора 1. В данном случае алгоритм получения данных должен быть другим.

Сначала процессору 0 нужно как-то выяснить, какой процессор содержит необходимые ему данные, и послать этому процессору сообщение с запросом копии данных. Затем процессор 0 блокируется до получения ответа. Когда процессор 1 получает сообщение, оно программно анализируется, после чего затребованные данные передаются обратно. Когда процессор 0 получает ответное сообщение, блокировка программно снимается, и процессор продолжает работу.

В мультикомпьютере для взаимодействия между процессорами часто используются примитивы `send` и `receive`. Поэтому программное обеспечение мультикомпьютера имеет более сложную структуру, чем программное обеспечение мультипроцессора. При этом основной проблемой становится правильное распределение данных и разумное их размещение. Это еще одно отличие мультикомпьютера от мультипроцессора, где размещение данных не влияет на правильность решения задачи, хотя может повлиять на производительность. Короче говоря, мультикомпьютер программировать гораздо сложнее, чем мультипроцессор.

Возникает вопрос: зачем вообще создавать мультикомпьютеры, если мультипроцессоры гораздо проще программировать? Ответ прост: создать большой мультикомпьютер проще и дешевле, чем мультипроцессор с таким же количеством процессоров. Реализация общей памяти, совместно используемой несколькими сотнями процессоров, — это весьма сложная задача, а разработать мультикомпьютер, содержащий 10 000 процессоров и более, довольно легко. Далее в этой главе мы рассмотрим мультикомпьютер с более чем 50 000 процессорами.

Таким образом, мы сталкиваемся с дилеммой: мультипроцессоры сложно разрабатывать, но легко программировать, а мультикомпьютеры легко строить, но трудно программировать. В результате постоянно предпринимаются попытки создания гибридных систем. Эти попытки привели к осознанию того факта, что совместную память можно реализовывать по-разному, причем каждый вариант будет иметь достоинства и недостатки. Практически все исследования в области параллельных компьютерных архитектур направлены на создание гибридных форм, которые сочетают в себе достоинства обеих систем. Здесь важно добиться **масштабируемости**, то есть разработать такую систему, которая будет продолжать исправно работать при добавлении все новых и новых процессоров.

Один из подходов основан на том, что современные компьютерные системы не монолитны, а имеют многоуровневую структуру. Это дает возможность реализовать общую память на любом из нескольких уровней, как показано на рис. 8.19. На рис. 8.19, *а* мы видим общую память, реализованную аппаратно, как в «настоящем» мультипроцессоре. В данной разработке имеется одна копия операционной системы с одним набором таблиц, в частности таблицей распределения памяти. Если процессу требуется больше памяти, он прерывает работу операционной системы, которая после этого начинает искать в таблице свободную страницу и отображает эту страницу на адресное пространство вызывающего процесса. Что касается операционной системы, имеется единая память, и операционная система следит, какая страница какому процессу принадлежит. Существует множество способов аппаратной реализации общей памяти.

Второй подход — использовать аппаратное обеспечение мультикомпьютера и операционную систему, которая будет моделировать общую память, предоставляя единое виртуальное адресное пространство, разбитое на страницы. При таком подходе получается **распределенная общая память** (Distributed Shared Memory, **DSM**), в которой каждая страница расположена в одном из модулей памяти (см. рис. 8.17, *а*), а каждая машина содержит собственную виртуальную память и собственные таблицы страниц [Li and Hudak, 1989]. Если процессор выполняет команду `LOAD` или `STORE`, обращаясь к странице, которой у него нет, происходит системное исключение. После этого операционная система находит нужную страницу и обращается к соответствующему процессору, чтобы тот выгрузил страницу из памяти и послал ее через внутреннюю коммуникационную сеть, по которой процессоры обмениваются сообщениями. Когда страница попадает процессу-получателю, она отображается на память, и выполнение прерванной команды возобновляется. По существу, операционная система просто получает недостающие страницы не с диска, а из памяти. При этом у пользователя создается впечатление, что машина имеет единую общую память. Далее в этой главе мы еще вернемся к распределенной общей памяти.

Третий подход — реализовать общую память программно пользовательской системой реального времени. При таком подходе абстракцию общей памяти

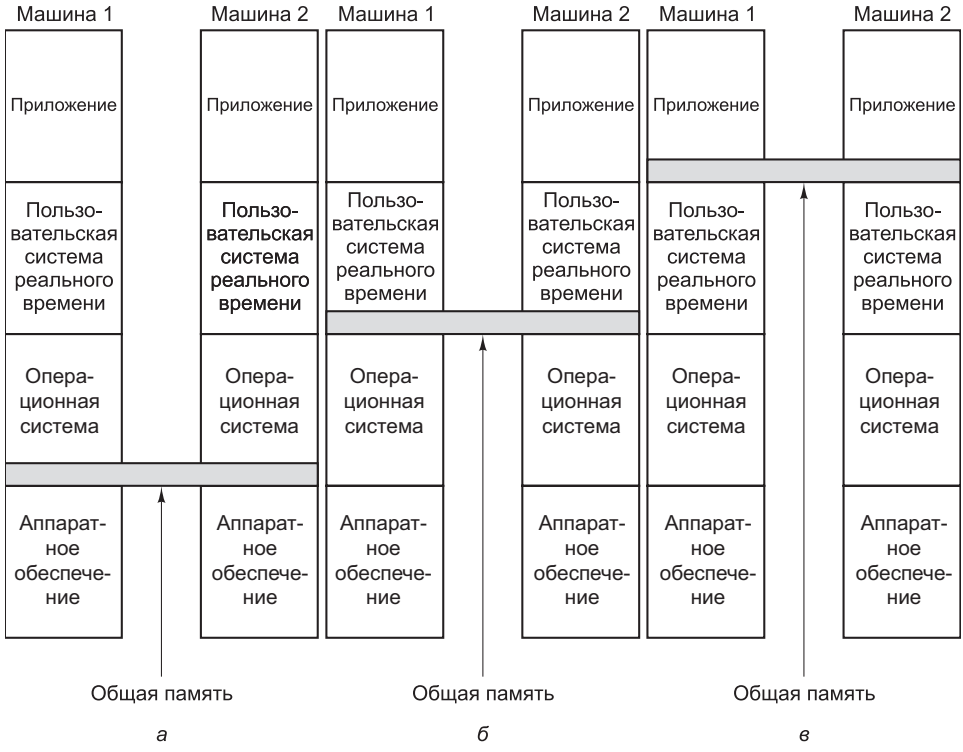


Рис. 8.19. Уровни, на которых можно реализовать общую память: аппаратная реализация (*а*); операционная система (*б*); программная реализация (*в*)

создает язык программирования, и эта абстракция реализуется компилятором (то есть модель общей памяти может зависеть от используемого языка программирования). Например, модель Linda основана на абстракции общего пространства кортежей (записей данных, содержащих наборы полей). Процессы любой машины могут взять кортеж из общего пространства или отправить его в общее пространство. Поскольку доступ к этому пространству полностью контролируется программно (системой реального времени Linda), никакой специальной аппаратной поддержки или особой операционной системы не требуется.

Другой пример общей памяти, реализованной пользовательской системой реального времени, — модель общих объектов данных в системе Orca. В модели Orca процессы совместно используют не кортежи, как в Linda, а базовые объекты, вызывая для них те или иные методы. Если метод изменяет внутреннее состояние объекта, система реального времени должна проследить, чтобы одновременно были изменены все копии этого объекта на всех машинах. И опять, поскольку объекты — чисто программная концепция, их можно реализовать с помощью системы реального времени без вмешательства операционной системы или аппаратного обеспечения. К моделям Linda и Orca мы еще вернемся в этой главе.

Классификация параллельных компьютерных систем

Многое можно сказать о программном обеспечении для параллельных компьютерных систем, однако сейчас мы должны вернуться к основной теме данной

главы — архитектуре таких систем. За прошедшие годы было предложено и построено множество видов параллельных компьютерных систем, поэтому хотелось бы их как-либо классифицировать. Это с разными результатами пытались делать многие исследователи [Flynn, 1972; Treleaven, 1985], но, к сожалению, хорошей классификации до сих пор нет. Чаще всего используют классификация Флинна, но даже она является в лучшем случае весьма приближенной (табл. 8.3).

Таблица 8.3. Классификация параллельных компьютерных систем по Флинну

Потоки команд	Потоки данных	Категория	Примеры
1	1	SISD	Классическая машина фон Неймана
1	Много	SIMD	Векторный суперкомпьютер, матричный процессор
Много	1	MISD	Не существует
Много	Много	MIMD	Мультипроцессор, мультикомпьютер

В основе классификации Флинна лежат понятия **потоков команд** и **потоков данных**. Поток команд соответствует счетчику команд. Система с n процессорами имеет n счетчиков команд и, следовательно, n потоков команд.

Поток данных состоит из набора операндов. Например, в системе прогнозирования погоды каждый из многочисленных датчиков может выдавать поток значений температуры, измеренной с регулярными интервалами.

Потоки команд и данных в какой-то степени независимы, поэтому существуют 4 комбинации таких потоков (см. табл. 8.3). SISD (Single Instruction stream Single Data stream — один поток команд с одним потоком данных) — это классическая последовательная компьютерная архитектура фон Неймана. Компьютер фон Неймана имеет один поток команд и один поток данных и в каждый момент времени может выполнять только одно действие. У машины, относящейся к категории SIMD (Single Instruction-stream Multiple Data-stream — один поток команд с несколькими потоками данных), имеется один блок управления, выдающий по одной команде, но при этом есть несколько АЛУ, которые могут обрабатывать несколько наборов данных одновременно. Прототип SIMD-машин — процессор ILLIAC IV (см. рис. 2.6). Хотя SIMD-машины не относятся к числу широко распространенных, в некоторых обычных компьютерах для обработки мультимедийных данных используются SIMD-команды. SSE-команды в процессорах Pentium относятся к категории SIMD-команд. В любом случае существует одна область, где идеи, почерпнутые из «мира SIMD», выходят на первый план, — это потоковые процессоры. Потоковые процессоры специально разработаны для обработки мультимедийных данных и в будущем они могут играть важную роль [Kapasi et al., 2003].

MISD (Multiple Instruction-stream Single Data-stream — несколько потоков команд с одним потоком данных) — довольно странная категория. Здесь несколько команд оперируют одним набором данных. Трудно сказать, существуют ли такие машины, хотя некоторые относят к категории MISD машины с конвейерами.

Последняя категория — MIMD (Multiple Instruction-stream Multiple Data-stream — несколько потоков команд с несколькими потоками данных). Здесь несколько независимых процессоров работают как часть большой системы. В эту категорию попадают большинство параллельных процессоров. И мультипроцессоры, и мультикомпьютеры относятся к MIMD-машинам.

Мы расширили классификацию Флинна (рис. 8.20). Категория SIMD-машин у нас разбита на две подгруппы. В первую подгруппу попадают многочисленные суперкомпьютеры и другие машины, которые оперируют векторами, выполняя одну и ту же операцию над каждым элементом вектора. Во вторую подгруппу попадают машины типа ILLIAC IV, в которых главный блок управления посылает команды нескольким независимым АЛУ.

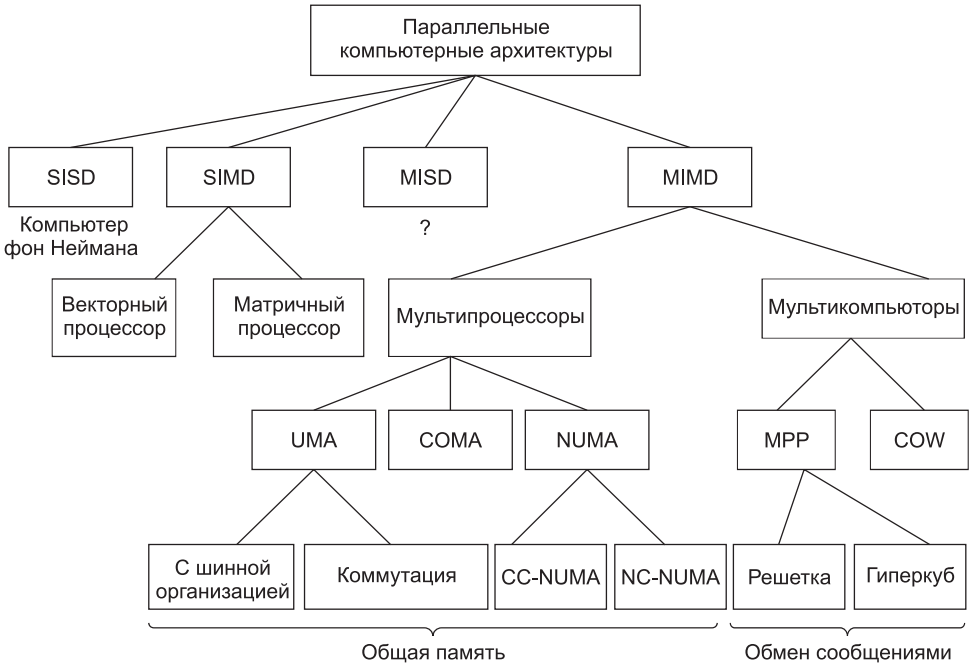


Рис. 8.20. Классификация компьютеров параллельного действия

В нашей классификации категория MIMD распалась на мультипроцессоры (машины с общей памятью) и мультикомпьютеры (машины с обменом сообщениями). Существует три типа мультипроцессоров. Они отличаются друг от друга механизмом доступа к общей памяти и называются **UMA** (Uniform Memory Access — **однородный доступ к памяти**), **NUMA** (NonUniform Memory Access — **неоднородный доступ к памяти**) и **COMA** (Cache Only Memory Access — **доступ только к кэш-памяти**). Такое разбиение на подкатегории имеет смысл, поскольку в больших мультипроцессорах память обычно делится на несколько модулей. В UMA-машинах каждый процессор имеет одно и то же время доступа к любому модулю памяти. Иными словами, каждое слово читается из памяти с той же скоростью, что и любое другое слово. Если это технически невозможно, самые быстрые обращения замедляются, чтобы соответствовать самым медленным, поэтому программист не заметит никакой разницы. Это и значит «однородный» доступ. Такая однородность делает производительность предсказуемой, а этот фактор очень важен для создания эффективных программ.

NUMA-машина, напротив, не обладает свойством однородности. Обычно у каждого процессора есть один из модулей памяти, который располагается к нему ближе, чем другие, поэтому обращения к этому модулю памяти происходят гораз-

до быстрее, чем к другим. В этом случае с точки зрения производительности очень важно, где окажутся программа и данные. Доступ к COMA-машинам тоже оказывается неоднородным, но по другой причине. Подробнее каждый из вариантов мы рассмотрим позднее, когда будем изучать соответствующие подкатегории.

Во вторую основную категорию MIMD-машин попадают мультикомпьютеры, которые в отличие от мультипроцессоров не имеют общей памяти на архитектурном уровне. Другими словами, операционная система процессора, входящего в состав мультикомпьютера, не сможет получить доступ к памяти другого процессора, просто выполнив команду LOAD. Процессору придется отправить сообщение и ждать ответа. Именно способность операционной системы считать слово из удаленного модуля памяти с помощью команды LOAD отличает мультипроцессоры от мультикомпьютеров. Как мы уже отмечали, хотя даже в мультикомпьютере пользовательские программы могут обращаться к другим модулям памяти с помощью команд LOAD и STORE, эта способность не подкреплена аппаратно, иллюзию создает операционная система. Разница незначительна, но очень важна. Так как мультикомпьютеры не имеют непосредственного доступа к удаленным модулям памяти, они иногда относят к категории **NORMA** (NO Remote Memory Access — **отсутствие удаленного доступа к памяти**).

Мультикомпьютеры тоже можно разделить на две дополнительные категории. К категории **MPP** (Massively Parallel Processor — процессор с массовым параллелизмом) относятся дорогостоящие суперкомпьютеры, которые состоят из большого количества процессоров, связанных высокоскоростной внутренней коммуникационной сетью. В качестве хорошо известного коммерческого примера можно назвать суперкомпьютер SP/3 компании IBM.

Вторая категория мультикомпьютеров включает обычные персональные компьютеры или рабочие станции (иногда смонтированные в стойки), которые связываются в соответствие с той или иной коммерческой коммуникационной технологией. С точки зрения логики, принципиальной разницы здесь нет, но мощный суперкомпьютер стоимостью в миллионы долларов безусловно используется иначе, чем собранная конечными пользователями компьютерная сеть, которая обходится во много раз дешевле любой MPP-машины. Эти «доморощенные» системы иногда называют **сетями рабочих станций** (Network Of Workstations, **NOW**), **кластерами рабочих станций** (Cluster Of Workstattions, **COW**) или просто **кластерами** (cluster).

Семантика памяти

Хотя во всех мультипроцессорах процессорам предоставляется образ общего единого адресного пространства, часто наряду с ним имеется множество модулей памяти, в каждом из которых хранится какой-то фрагмент физической памяти. Процессоры и модули памяти соединяются сложной коммуникационной сетью (см. выше раздел «Внутрипроцессорная многопоточность» этой главы). Несколько процессоров могут попытаться считать слово из памяти в то же время, когда другие процессоры будут пытаться его записать; сообщения могут доставляться не в том порядке, в котором они были отправлены. Добавим к этим проблемам существование многочисленных копий некоторых фрагментов памяти (например, в кэш-памяти), и в результате мы придем к хаосу, если не принять серьезные контрмеры. В этом подразделе мы выясним, что в действительности

представляет собой общая память и как при таких обстоятельствах можно разумно использовать модули памяти.

Семантику памяти можно рассматривать как контракт между программным и аппаратным обеспечением памяти [Adve and Hill, 1990]. Если программное обеспечение соглашается следовать определенным правилам, то память соглашается выдавать определенные результаты. Основная проблема здесь — сами правила, которые называются **моделями состоятельности**. Было предложено и разработано множество таких правил [Sorin et al., 2011].

Чтобы представить себе суть проблемы, предположим, что процессор 0 записывает значение 1 в какое-то слово памяти, а немного позже процессор 1 записывает значение 2 в то же самое слово. Процессор 2 считывает это слово и получает значение 1. Должен ли владелец компьютера обратиться после этого в бюро ремонта? Это зависит от того, что обещано в контракте.

Строгая состоятельность

Самая простая модель — модель **строгой состоятельности**. В такой модели при любом считывании из адреса *x* всегда возвращается значение самой последней записи в *x*. Программистам очень нравится эта модель, но ее можно реализовать на практике только следующим образом: должен быть единственный модуль памяти, просто обслуживающий все запросы по мере их поступления (первым поступил — первым обработан), кэширование и дублирование данных не допускаются. К несчастью, этот подход значительно затормозил бы работу памяти, поэтому вряд ли может рассматриваться в качестве серьезного предложения.

Секвенциальная состоятельность

Следующей мы рассмотрим модель **секвенциальной состоятельности** [Lamport, 1979]. В соответствии с этой моделью при наличии нескольких запросов на чтение и запись порядок обработки запросов определяется аппаратно, но при этом все процессоры воспринимают один и тот же порядок.

Рассмотрим пример. Предположим, процессор 1 записывает значение 100 в слово *x*, а через 1 нс процессор 2 записывает туда же значение 200. А теперь предположим, что через 1 нс после начала второй операции записи (процесс записи еще не закончен) два других процессора, 3 и 4, считывают слово *x* по два раза (рис. 8.21).

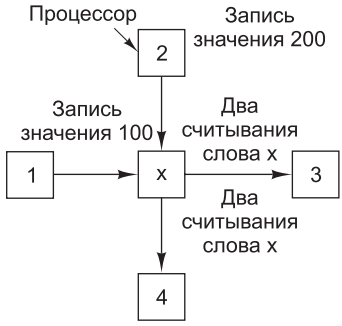


Рис. 8.21. Два процессора записывают, а другие два процессора считывают одно и то же слово из общей памяти

Возможные варианты очередности шести событий представлены в табл. 8.4.

Таблица 8.4. Возможные варианты очередности событий согласно рис. 8.21

Вариант 1	Вариант 2	Вариант 3
Запись значения 100	Запись значения 100	Запись значения 200
Запись значения 200	Чтение значения 100 процессором 3	Чтение значения 200 процессором 4
Чтение значения 200 процессором 3	Запись значения 200	Запись значения 100
Чтение значения 200 процессором 3	Чтение значения 200 процессором 4	Чтение значения 100 процессором 3
Чтение значения 200 процессором 4	Чтение значения 200 процессором 3	Чтение значения 100 процессором 4
Чтение значения 200 процессором 4	Чтение значения 200 процессором 4	Чтение значения 100 процессором 3

В первом варианте оба процессора получают значение 200 в каждой из двух операций считывания. Во втором варианте процессор 3 получает значения 100 и 200, а процессор 4 — оба раза по 200. В третьем варианте процессор 3 получает два раза по 100, а процессор 4 — значения 200 и 100. Все эти варианты допустимы, как и некоторые другие, которые здесь не показаны. «Единственно правильного» значения не существует.

Память, построенная в соответствие с моделью секвенциальной состоятельности, никогда не позволит процессору 3 получить значения 100 и 200, если процессор 4 получает значения 200 и 100. Если бы это произошло, с точки зрения процессора 3 это бы означало, что запись значения 100 процессором 1 завершилась раньше записи значения 200, которую осуществляет процессор 2. Это вполне возможно. Но с точки зрения процессора 4 это также значит, что запись процессором 2 числа 200 завершилась до записи процессором 1 числа 100. Сам по себе такой результат тоже возможен, но он противоречит первому результату. Секвенциальная состоятельность гарантирует единую глобальную (воспринимаемую всеми процессорами) последовательность операций записи. Если с точки зрения процессора 3 первым записывается значение 100, процессор 4 должен «видеть» тот же самое.

Хотя правила секвенциальной состоятельности не выглядят столь «жесткими», как правила строгой, эта модель тоже очень полезна. Даже если несколько событий совершаются одновременно, считается, что на самом деле они происходят в определенном порядке (который может выбираться произвольно), и все процессоры воспринимают именно этот порядок. Хотя такое положение дел и кажется очевидным, далее мы рассмотрим некоторые модели состоятельности, которые не гарантируют подобного порядка.

Процессорная состоятельность

Процессорная состоятельность — не слишком строгая модель, но зато ее легче реализовать на больших мультипроцессорах [Goodman, 1989]. Она имеет два свойства.

1. Все процессоры видят операции записи любого процессора в том порядке, в котором эти операции выполняются.
2. Все процессоры видят все операции записи в любое слово памяти в одном и том же порядке.

Оба этих пункта очень важны. В первом пункте говорится, что если процессор 1 начинает запись значений 1A, 1B и 1C в какое-либо место в памяти именно в таком порядке, то все другие процессоры увидят эти записи в том же порядке. Иными словами, никогда не произойдет такого, чтобы какой-либо процессор сначала увидел значение 1B, а затем значение 1A. Второй пункт нужен, чтобы каждое слово в памяти имело определенное и недвусмысленное значение после того, как процессор совершил несколько записей в это слово, а затем остановился. Все должны видеть одно и то же последнее значение.

Даже при таких ограничениях у разработчика есть много возможностей. Посмотрим, что произойдет, если процессор 2 начинает три операции записи значений 2A, 2B и 2C одновременно с тремя операциями записи процессора 1. Другие процессоры, которые заняты считыванием слов из памяти, увидят какую-либо последовательность из шести операций записи, например 1A, 1B, 2A, 2B, 1C, 2C или 2A, 1A, 2B, 2C, 1B, 1C и т. п. При процессорной состоятельности не гарантируется, что каждый процессор видит один и тот же порядок (в отличие от секвенциальной состоятельности). Вполне может быть так, что одни процессоры воспринимают порядок 1A, 1B, 2A, 2B, 1C, 2C, другие — 2A, 1A, 2B, 2C, 1B, 1C, третьи — еще какой-нибудь вариант. Единственное, что гарантируется абсолютно точно, — ни один процессор не увидит последовательность, в которой сначала выполняется операция 1B, а затем — 1A. Порядок, в котором выполняются обращения одного и того же процессора, остается одинаковым для всех наблюдателей.

Следует заметить, что некоторые авторы определяют процессорную состоятельность иначе и не требуют выполнения второго условия.

Слабая состоятельность

В модели **слабой состоятельности** не гарантируется, что операции записи, произведенные одним процессором, будут восприниматься другими в том же порядке [Dubois et al., 1986]. Один процессор может увидеть сначала операцию 1A, а потом 1B, другой — сначала 1B, потом 1A. Чтобы внести порядок в этот хаос, должны иметься переменные синхронизации памяти или поддерживаться операция синхронизации памяти. При синхронизации все незаконченные операции записи завершаются, и ни одна новая операция не может начаться, пока не будут завершены все предыдущие записи и не завершится сама синхронизация. Синхронизация приводит память в устойчивое состояние, когда не остается никаких незавершенных операций. Сами операции синхронизации являются секвенциально состоятельными, то есть если они инициируются несколькими процессорами, выбирается определенный порядок их выполнения, причем все процессоры воспринимают один и тот же порядок.

При слабой состоятельности время разделяется на строго последовательные периоды, разделенные операциями синхронизации (рис. 8.22). Никакого особого порядка для операций записи 1A и 1B не гарантируется, и разные процессоры могут воспринимать их по-разному, то есть с точки зрения одного процессора сначала может выполняться операция 1A, а затем 1B, а с точки зрения другого — сначала 1B, а затем 1A. Такая ситуация допустима. Однако для всех процессоров операция 1B выполнена раньше 1C, поскольку записи 1C, 2B, 3A, 3B могли начаться только после того, как в ходе первой операции синхронизации завершились записи 1A, 1B и 2A. Таким образом, с помощью операций синхронизации

программно можно вносить некий порядок в последовательность событий, хотя это занимает некоторое время, поскольку требует очистки конвейера памяти. Слишком частое выполнение таких операций создает проблемы.

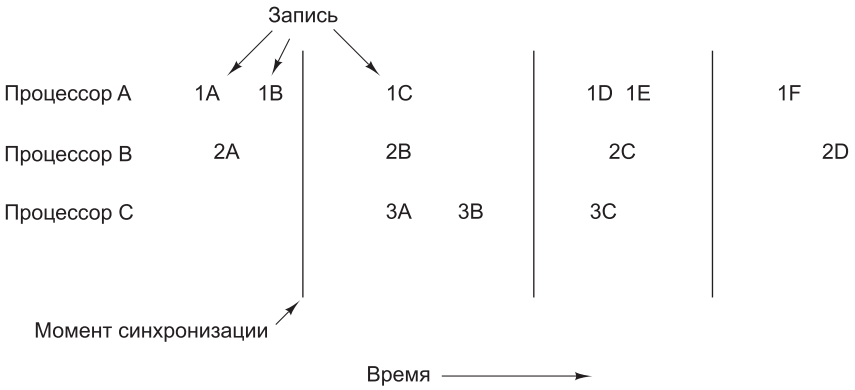


Рис. 8.22. В слабо состоятельной памяти периодически выполняются операции синхронизации

Свободная состоятельность

Слабая состоятельность — не очень эффективная модель, поскольку требует завершения всех операций с памятью и задерживает выполнение новых операций до тех пор, пока не завершены предыдущие. В модели **свободной состоятельности** дела обстоят гораздо лучше, поскольку здесь используется нечто похожее на критические секции программы [Gharachorloo et al., 1990]. Идея состоит в следующем. Если процесс выходит за пределы критической области, это не значит, что все записи должны немедленно завершиться. Требуется только, чтобы все записи были завершены до того, как какой-нибудь процесс снова войдет в эту критическую область.

В этой модели операция синхронизации разделяется на две разные операции. Чтобы считать или записать совместно используемую переменную, процессор (то есть его программное обеспечение) сначала должен выполнить операцию `acquire` с переменной синхронизации, что позволит ему получить монопольный доступ к общим данным. Далее процессор может делать с этими данными все, что ему требуется (считывать или записывать), а по завершении он должен выполнить операцию `release` с переменной синхронизации, чтобы показать, что он завершил работу. Операция `release` не требует завершения незаконченных записей, но сама она не может завершиться, пока не закончатся все ранее начатые операции записи. Более того, новые операции с памятью могут начинаться сразу же.

Когда начинается следующая операция `acquire`, производится проверка, все ли предыдущие операции `release` завершены. Если нет, то операция `acquire` задерживается до тех пор, пока это не будут сделано (а перед тем, как завершатся все операции `release`, должны быть завершены все операции записи). Таким образом, если следующая операция `acquire` выполняется через достаточно длительный промежуток времени после последней операции `release`, ей не нужно ждать, и она может войти в критическую область без задержки. Если операция `acquire` выполняется через небольшой промежуток времени после операции `release`, она (и все команды, которые должны выполняться следом) ожидает завершения всех

операций `release`. Это гарантирует, что все переменные в критической области будут обновлены. Такая модель немного сложнее, чем модель слабой состоятельности, но она имеет существенное преимущество: здесь не нужно задерживать выполнение команд так часто, как в модели слабой состоятельности.

Вопрос о состоятельности памяти нельзя считать окончательно решенным. Исследователи до сих пор предлагают новые модели [Naeem et al., 2011; Sorin et al., 2011; Tu et al., 2010].

UMA-мультипроцессоры в симметричных мультипроцессорных архитектурах

Самые простые мультипроцессоры имеют всего одну шину (рис. 8.23, *а*). Два или более процессоров и один или несколько модулей памяти используют эту шину для взаимодействия. Если процессору нужно считать слово из памяти, он сначала проверяет, свободна ли шина. Если шина свободна, процессор помещает адрес нужного слова на шину, устанавливает несколько управляющих сигналов и ждет, когда память поместит на шину запрошенное слово.

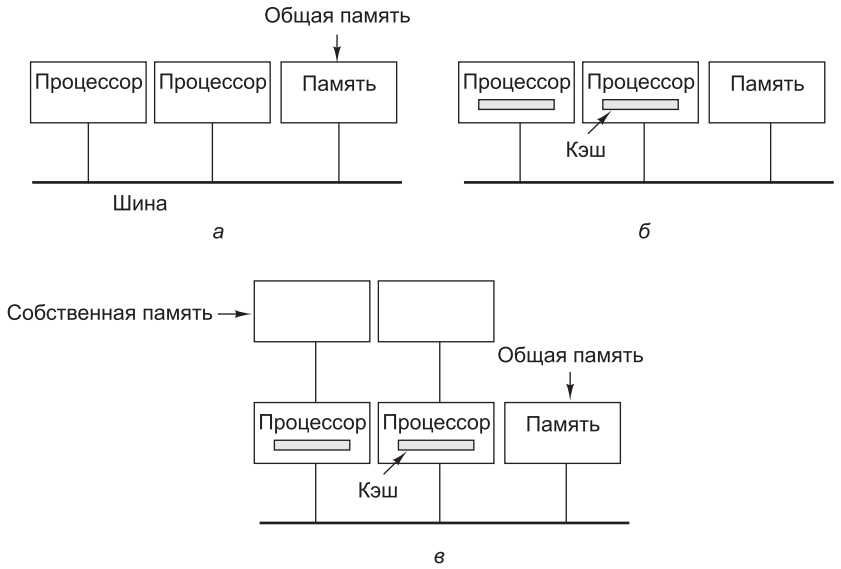


Рис. 8.23. Три варианта мультипроцессора на одной шине: без кэш-памяти (*а*); с кэш-памятью (*б*); с кэш-памятью и отдельными модулями локальной памяти (*в*)

Если шина занята, процессор просто ждет, когда она освободится. С этой схемой связана одна проблема. При наличии двух или трех процессоров доступ к шине регулировать не сложно, трудности возникают, когда процессоров 32 или 64. Производительность системы в этом случае полностью определяется пропускной способностью шины, и многим процессорам большую часть времени приходится простаивать.

Чтобы разрешить проблему, нужно добавить к каждому процессору кэш-память, как показано на рис. 8.23, *б*. Кэш-память может находиться внутри микросхемы процессора, рядом с микросхемой процессора, на плате процессора.

Допустимы любые комбинации этих вариантов. Поскольку в этом случае считывать многие слова можно будет из кэша, трафик на шине снизится, и система сможет обслуживать большее количество процессоров. Таким образом, кэширование дает в данном случае значительный эффект. Однако, как мы вскоре увидим, согласование содержимого кэшей — задача далеко не тривиальная.

В следующей схеме каждый процессор имеет не только кэш, но и собственную локальную память, к которой он получает доступ через выделенную локальную шину (рис. 8.23, *в*). Чтобы оптимально задействовать эту конфигурацию, компилятор должен поместить в локальные модули памяти весь программный код, строки, константы и другие данные, предназначенные только для чтения, а также стеки и локальные переменные. Общая память потребуется только для хранения совместно используемых переменных. В большинстве случаев такое разумное распределение значительно снижает интенсивность трафика на шине и не требует активного содействия со стороны компилятора.

Согласованность кэшей

Безусловно, упомянутые аспекты производительности важны, но мы обошли стороной одну фундаментальную проблему. Предположим, что память секвенциально состоятельна. Что происходит, когда процессор 1 содержит в своем кэше некую строку, а процессор 2 пытается считать слово, соответствующее той же строке кэша? При отсутствии специальных правил процессор 2 получит копию этой строки в свой кэш. В принципе, двойное кэширование одной и той же строки вполне приемлемо. А теперь предположим, что процессор 1 изменяет строку, и сразу после этого процессор 2 считывает копию этой строки из своей кэш-памяти. В результате он получает **устаревшие данные**, нарушая контракт между программным обеспечением и памятью. Ни к чему хорошему для программы, выполняемой процессором 2, это не приведет.

Данная проблема, которая носит название **проблемы согласованности кэшей**, очень важна. Если ее не разрешить, нельзя будет использовать кэш-память, и число мультипроцессоров, подсоединенных к одной шине, придется сократить до двух-трех. Специалистами было предложено множество различных решений (см., например, [Goodman, 1983; Papamarcos and Patel, 1984]). Хотя все эти алгоритмы, называемые **протоколами согласования кэшей**, в некоторых деталях различаются, все они не допускают одновременного появления разных версий одной и той же строки в двух или более кэшах.

Во всех решениях контроллер кэш-памяти разрабатывается так, чтобы кэш мог обеспечивать мониторинг запросов, идущих по шине от других процессоров и других кэшей, в каждом конкретном случае предпринимая те или иные действия. Указанное устройство получило название **следящего кэша** (snooping cache), поскольку кэш как бы «следит» за шиной. Набор правил, которым следуют кэш, процессоры и основная память, чтобы предотвратить появление различных версий данных в нескольких кэшах, и называют протоколом согласования кэшей. Единицей передачи и хранения для кэша является **строка кэша**. Обычно длина строки кэша составляет 32 или 64 байт.

Самый простой протокол согласования кэшей называется **сквозной записью** (write through). Чтобы лучше понять механизм работы этого протокола, рассмотрим 4 случая, перечисленные в табл. 8.5. Если процессор пытается считать слово, которого нет в кэш-памяти, контроллер кэш-памяти загружает в кэш

строку, содержащую это слово. Строку предоставляет основная память, которая в этом протоколе всегда должна хранить обновленные данные. В дальнейшем информация может считываться из кэша.

Таблица 8.5. Сквозная запись (пустые графы означают, что никакого действия не происходит)

Действие	Локальный запрос	Удаленный запрос
Кэш-промах чтения	Вызов данных из памяти	
Кэш-попадание чтения	Использование данных из локального кэша	
Кэш-промах записи	Обновление данных в памяти	
Кэш-попадание записи	Обновление кэша и памяти	Объявление элемента в кэше недействительным

В случае кэш-промаха записи модифицированное слово записывается в основную память. Строка, содержащая нужное слово, *не* загружается в кэш. В случае кэш-попадания записи кэш обновляется, а слово к тому же записывается в основную память. Суть протокола состоит в том, что в результате всех операций записи записываемое слово обязательно проходит через основную память, чтобы данные в основной памяти всегда были «свежими».

Рассмотрим все эти действия снова, но теперь с позиции следящего кэша (крайняя правая колонка в табл. 8.5). Назовем кэш, который выполняет действия, кэшем 1, а следящий кэш — кэшем 2. Если происходит кэш-промах чтения кэша 1, кэш 1 обращается к шине, чтобы получить нужную строку из основной памяти. Кэш 2 видит это, но ничего не делает. Если же происходит кэш-попадание чтения кэша 1 (то есть нужная строка уже содержится в кэше 1), обращение к шине не выполняется, поэтому кэш 2 ничего не знает о кэш-попаданиях чтения кэша 1.

Процесс записи более интересен. Если процессор 1 записывает слово, кэш 1 обращается к шине как в случае кэш-промаха, так и в случае кэш-попадания. При любой записи кэш 2 проверяет наличие записываемого слова у себя. Если слово отсутствует, кэш 2 рассматривает это как кэш-промах удаленной памяти и ничего не делает. (Отметим, что согласно табл. 8.5, кэш-промах удаленной памяти означает, что слово отсутствует в следящем кэше, а есть оно в кэше-инициаторе или нет, значения не имеет. Таким образом, один и тот же запрос может дать локальное кэш-попадание и кэш-промах для следящего кэша, и наоборот.)

А теперь предположим, что кэш 1 записывает слово, которое *есть* в кэше 2. Если кэш 2 ничего не предпримет, он будет содержать устаревшие данные, поэтому кэш 2 помечает как недействительный тот элемент кэш-памяти, который содержит измененное слово. В результате элемент просто удаляется из кэша. Поскольку все кэши следят за всеми обращениями к шине, запись любого слова ведет к обновлению его в кэше-инициаторе и в основной памяти, а также к удалению его из всех других кэшей. Таким образом предотвращается появление несогласованных версий.

Естественно, процессор кэша 2 вправе прочитать то же самое слово на следующем цикле. В этом случае кэш 2 получает слово из основной памяти, которая уже обновилась. В этот момент кэш 1, кэш 2 и основная память содержат идентичные копии этого слова. Если же какой-нибудь процессор произведет запись, то кэши других процессоров будут очищены, а основная память опять обновится.

Возможны различные вариации этого основного протокола. Например, в случае кэш-попадания записи следящий кэш обычно объявляет недействительным элемент, содержащий записываемое слово. Однако вместо того чтобы объявлять слово недействительным, можно принять новое значение и обновить кэш. По существу, обновление кэша — это то же самое, что объявление элемента недействительным с последующим считыванием нужного слова из памяти. Во всех протоколах кэширования нужно делать выбор между **стратегией обновления** и **стратегией объявления данных недействительными**. Эти протоколы работают по-разному при разной нагрузке. Сообщения обновления несут полезную нагрузку, и, следовательно, они больше по размеру, чем сообщения о недействительности данных, но зато они могут предотвратить последующие кэш-промахи.

Еще один вариант — загрузка следящего кэша при кэш-промахах записи. Такая загрузка никак не сказывается на правильности алгоритма; она влияет только на производительность. Возникает вопрос: какова вероятность, что только что записанное слово вскоре опять будет записано? Когда вероятность высока, можно говорить в пользу загрузки кэша при кэш-промахах записи (**политика заполнения по записи**). Когда вероятность мала, то в случае кэш-промаха записи лучше не обновлять кэш-память. Если данное слово вскоре должно быть *считано*, после кэш-промаха чтения оно все равно окажется загруженным, поэтому нет смысла загружать его при кэш-промахе записи.

Как и большинство простых решений, это решение не слишком эффективно. Каждая операция записи требует передачи данных в основную память по шине, и при большом количестве процессоров шина становится «узким местом». Поэтому были разработаны другие протоколы. Все они имеют одно общее свойство: не каждая операция записи ведет к записи непосредственно в основную память. Вместо этого при изменении строки кэша внутри кэша устанавливается особый бит, который указывает, что строка в кэше правильная, а в памяти — нет. Хотя в конечном итоге данную строку все равно придется записать в память, вполне вероятно, что это произойдет после выполнения еще нескольких операций записи. Такой тип протокола называется **протоколом отложенной записи**.

Протокол MESI

Одним из популярных протоколов отложенной записи является протокол **MESI** (Invalid, Shared, Exclusive, Modified — недействительный, разделяемый, эксклюзивный, модифицированный), названный так по первым буквам четырех возможных состояний элементов кэша [Papamarcos and Patel, 1984]. В его основе лежит более ранний **протокол однократной записи** [Goodman, 1983]. Протокол MESI используется в Core i7 и других процессорах для слежения за шиной. В соответствии с этим протоколом каждый элемент кэша может находиться в одном из следующих четырех состояний:

- недействительный — элемент кэша содержит недействительные данные;
- разделяемый — элемент может храниться в нескольких кэшах, память обновлена;
- эксклюзивный — элемент находится только в данном кэше (ни в каких других кэшах его нет), память обновлена;
- модифицированный — элемент действителен, основная память недействительна, копий элемента не существует.

При загрузке процессора все элементы кэша помечаются как недействительные. При первом считывании из основной памяти нужная строка вызывается в кэш данного процессора и помечается как эксклюзивная, поскольку это — единственная кэшированная копия (рис. 8.24, *а*). При последующих считываниях процессор использует эту строку, не обращаясь к шине. Другой процессор может вызвать ту же строку и поместить ее в кэш. В этом случае первый держатель

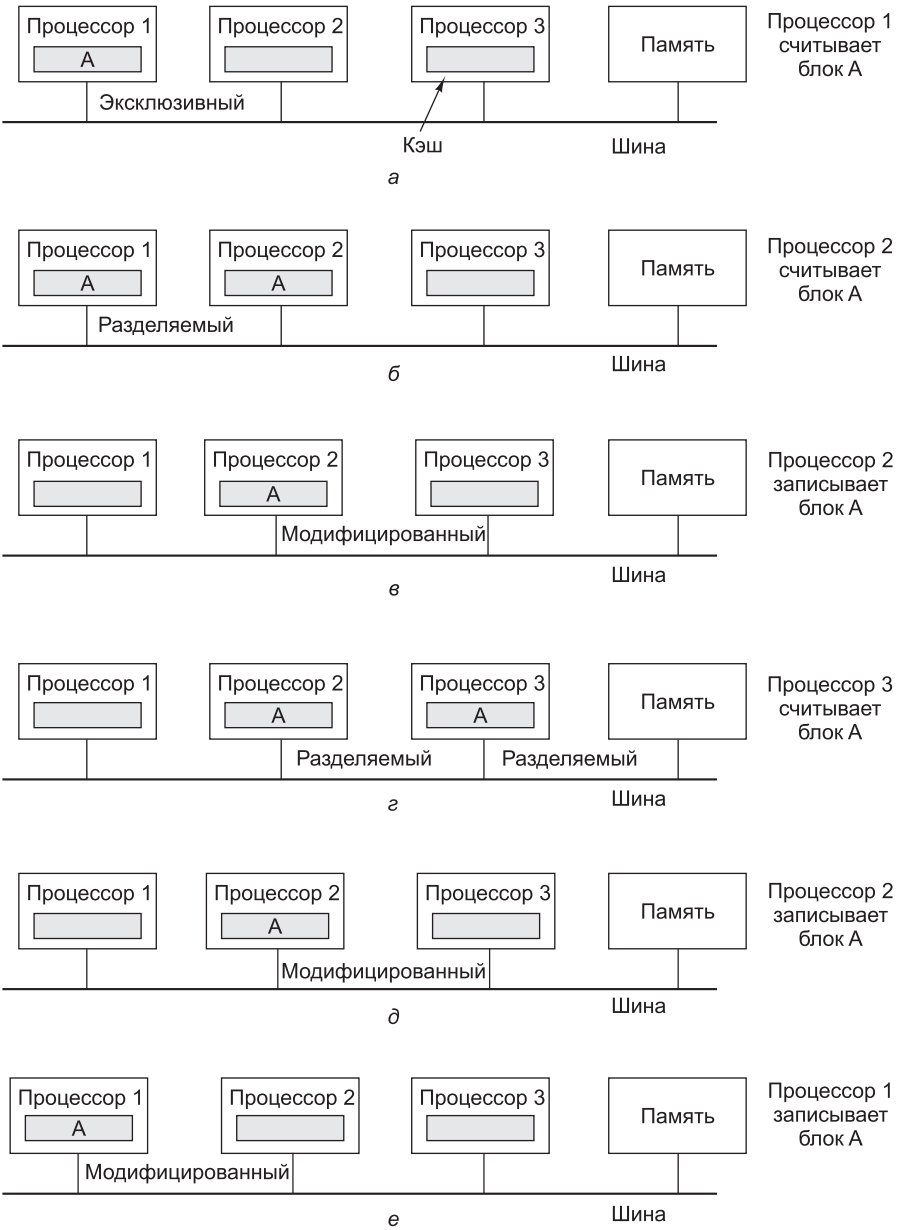


Рис. 8.24. Иллюстрация протокола MESI

строки (процессор 1) благодаря слежению узнает, что он уже не единственный держатель строки, и объявляет по шине, что у него копия. Обе копии помечаются как разделяемые (рис. 8.24, *б*). Другими словами, состояние «разделяемая» означает, что память обновлена и строка после чтения находится в одном или нескольких кэшах. При последующих чтениях разделяемой строки процессор не использует шину и не меняет состояние строки.

Посмотрим теперь, что происходит, когда процессор 2 выполняет запись в разделяемую строку кэша. В этом случае процессор помещает на шину специальный сигнал, сообщая всем другим процессорам о том, что их копии более недействительны, при этом копия в кэше самого процессора 2 переходит в состояние «модифицирована» (рис. 8.24, *в*). В память эта строка не записывается. Отметьте, что если указанная строка является эксклюзивной, никакого сигнала о недействительности на шину передавать не нужно, поскольку других копий этой строки не существует.

Далее разберем, что происходит при считывании этой строки процессором 3. Процессор 2, который в данный момент является держателем строки, знает, что копия в памяти недействительна, поэтому он передает на шину сигнал о том, чтобы процессор 3 ждал, пока он запишет строку обратно в память. Сразу после записи процессор 3 вызывает из памяти копию только что записанной строки, и в обоих кэшах строка помечается как разделяемая (рис. 8.24, *г*). Когда затем процессор 2 снова записывает эту строку (рис. 8.24, *в*), ее копия в кэше процессора 3 становится недействительной.

Наконец, пусть теперь слово в строке записывает процессор 1. Процессор 2 видит, что делается попытка записи, и выставляет на шину сигнал, который сообщает процессору 1, чтобы тот подождал, пока процессор 2 запишет свою строку в память. После окончания записи процессор 2 помечает собственную копию строки как недействительную, поскольку знает, что другой процессор собирается ее изменить. Возникает ситуация, в которой процессор выполняет запись в некэшированную строку. Если применяется политика заполнения по записи, строка будет загружена в кэш и помечена как модифицированная (рис. 8.24, *е*). Если политика заполнения по записи не применяется, запись происходит непосредственно в память, и строка вообще нигде не кэшируется.

UMA-мультипроцессоры с перекрестной коммутацией

Из-за наличия всего одной шины в UMA-мультипроцессоре даже после оптимизации не может быть больше 16 или 32 процессоров. Чтобы процессоров стало больше, требуется другой тип коммуникационной сети. Самая простая схема соединения n процессоров с k блоками памяти — **перекрестная коммутация** (рис. 8.25). Перекрестная коммутация на протяжении многих десятилетий используется в телефонных коммутаторах, позволяющих произвольным образом связывать группы входящих и исходящих линий.

На каждом пересечении горизонтальной (входящей) и вертикальной (исходящей) линии находится **коммутационный узел** (crosspoint), который можно открыть или закрыть в зависимости от того, нужно соединить горизонтальную и вертикальную линии или нет. На рис. 8.25, *а* мы видим, что три узла закрыты, благодаря чему одновременно устанавливается связь между следующими парами процессор-память (001, 000), (101, 101) и (110, 010). Возможны и другие комби-

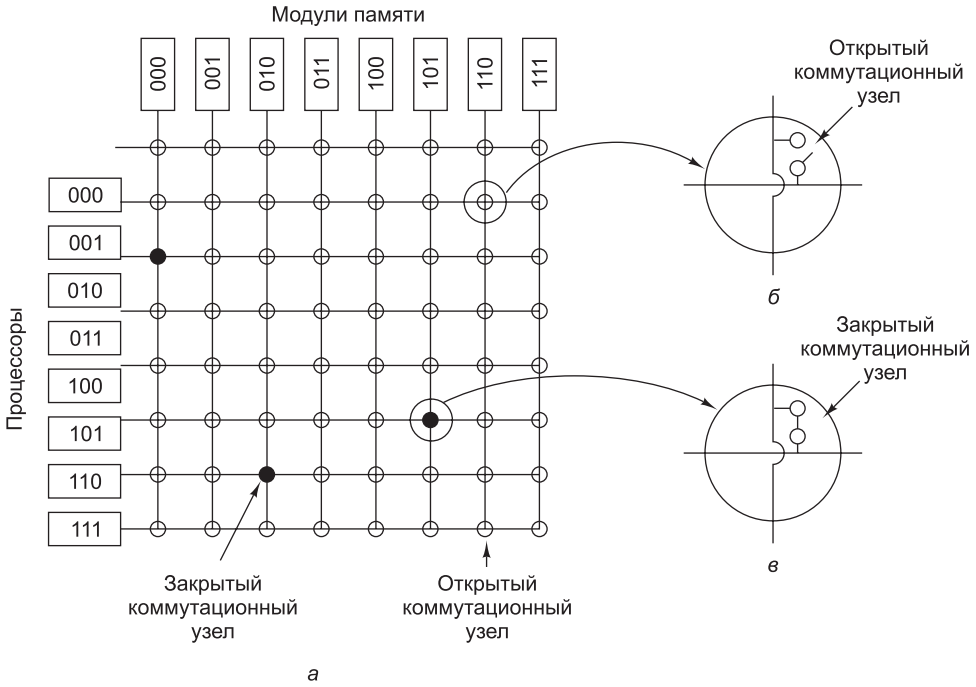


Рис. 8.25. Перекрестная коммутация 8 × 8 (*а*); открытый узел (*б*); закрытый узел (*в*)

нации. Число комбинаций равно числу вариантов расстановки восьми ладей на шахматной доске так, чтобы ни одна из них не находилась под боем другой.

Одним из самых полезных свойств сети с перекрестной коммутацией является то, что она **неблокирующая**. Это означает, что процессор всегда сможет соединиться с нужным модулем памяти, даже если некоторые линии или узлы уже заняты (предполагается, что сам модуль памяти доступен). Более того, никакого предварительного планирования не требуется. Даже если уже установлено семь произвольных соединений, всегда можно соединить оставшийся процессор с оставшимся модулем памяти. Далее мы рассмотрим схемы взаимного соединения, которые не обладают такой возможностью.

Одним из худших свойств перекрестной коммутации является то, что число узлов растет со скоростью n^2. Для средних по размеру систем перекрестная коммутация является хорошим решением, и далее в этой главе мы рассмотрим одно из таких решений — NUMA-мультипроцессор Sun Fire E25K. Однако для 1000 процессоров и 1000 модулей памяти понадобится миллион узлов, что неприемлемо. Необходимо нечто совершенно иное.

UMA-мультипроцессоры с многоступенчатой коммутацией

В основе этого «совершенно иного» лежит небольшой коммутатор 2 × 2 (рис. 8.26, *а*) с двумя входами и двумя выходами. Сообщения, приходящие на любую из входных линий, могут переключаться на любую выходную линию. В нашем примере сообщения будут содержать до четырех частей (рис. 8.26, *б*). Поле модуля показывает, какой модуль памяти запрашивается. Поле адреса

определяет адрес в этом модуле памяти. В поле кода операции указывается одна из доступных операций, например `READ` или `WRITE`. Наконец, дополнительное поле значения может содержать операнд, например 32-разрядное слово, которое нужно записать при выполнении операции `WRITE`. Коммутатор проверяет поле модуля и с его помощью определяет, через какую выходную линию нужно отправить сообщение: через *X* или через *Y*.

Рис. 8.26. Коммутатор 2 × 2 (*а*); формат сообщения (*б*)

Наши коммутаторы 2 × 2 можно компоновать различными способами и получать **сети с многоступенчатой коммутацией**. Один из возможных вариантов — **сеть omega** (рис. 8.27). Здесь 8 процессоров соединены с 8 модулями памяти через 12 коммутаторов. Для n процессоров и n модулей памяти понадобится $\log_2 n$ ступеней и $n/2$ коммутаторов на каждую ступень, то есть всего $(n/2)\log_2 n$ коммутаторов, что намного меньше, чем n^2 коммутационных узлов при перекрестной коммутации, особенно для больших значений n.

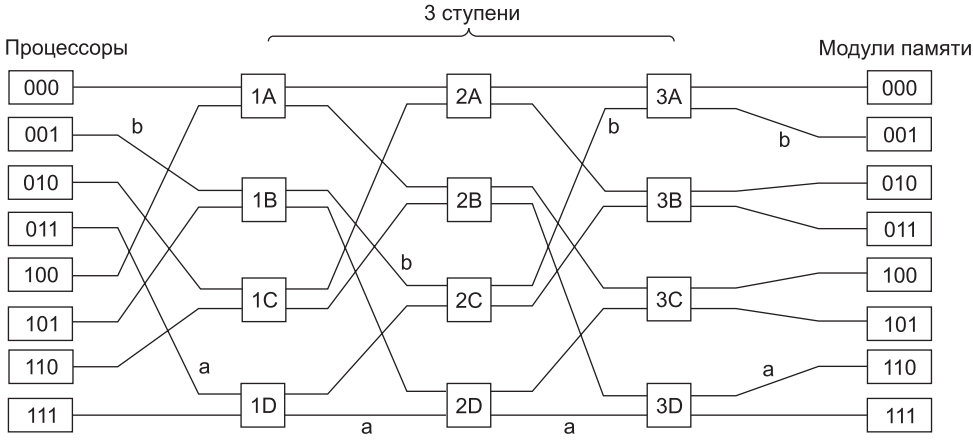


Рис. 8.27. Сеть с многоступенчатой коммутацией omega

Схему разводки проводов сети omega часто называют **полной перетасовкой** (perfect shuffle), поскольку смешение сигналов на каждой ступени напоминает тасование колоды карт. Чтобы понять, как работает сеть omega, предположим, что процессору 011 нужно считать слово из модуля памяти 110. Процессор посылает сообщение `READ`, чтобы переключить коммутатор 1D, который содержит 110 в поле модуля. Коммутатор берет первый (то есть крайний левый) бит от 110 и по нему узнает направление (0 указывает на верхний выход, 1 — на нижний). Поскольку в данном случае этот бит равен 1, сообщение отправляется через нижний выход к коммутатору 2D.

Все коммутаторы второй ступени, включая 2D, для определения направления используют второй бит. В данном случае он равен 1, поэтому сообщение отправ-

ляется через нижний выход к коммутатору 3D, который проверяет третий бит. Он равен 0, следовательно, сообщение проходит через верхний выход и прибывает в модуль памяти 110, чего мы и добивались. Путь, пройденный сообщением, обозначен на рис. 8.27 буквой *a*.

По мере прохождения через сеть сообщение поочередно перестает нуждаться во всех битах номера модуля, начиная с самого левого. Их можно использовать для записи номеров входных линий, чтобы было известно, по какому пути посылать ответ. Для пути *a* входные линии — это 0 (верхний вход в 1D), 1 (нижний вход в 2D) и 1 (нижний вход в 3D) соответственно. Таким образом, при отправке ответа тоже используется последовательность 011, только прочтенная справа налево.

Пусть в то время, как все это происходит, процессор 001 тоже решает записать слово в модуль памяти 001. Здесь происходит аналогичный процесс. Сообщение отправляется через верхний, верхний и нижний выходы соответственно. На рис. 8.27 этот путь отмечен буквой *b*. Когда сообщение пребывает в пункт назначения, в его поле модуля содержится последовательность 001, показывая путь, который прошло сообщение. Поскольку эти два запроса проходят через разные коммутаторы, линии и модули памяти, они могут выполняться параллельно.

А теперь посмотрим, что произошло бы, если бы процессору 000 понадобился доступ к модулю памяти 000. Его запрос вступил бы в конфликт с запросом процессора 001 на коммутаторе 3A, и одному из них пришлось бы ждать. То есть в отличие от сети с перекрестной коммутацией, сеть omega — это **блокирующая сеть**. Одновременно может передаваться не всякий набор запросов. Конфликты могут возникать как между запросами (при использовании одной и той же линии или одного и того же коммутатора), так и между запросами (*к* памяти) и ответами (*из* памяти).

Совершенно очевидно, что обращения к памяти желательно равномерно распределять по модулям памяти. Один из возможных способов — использовать младшие биты в качестве номера модуля. Рассмотрим адресное пространство с побайтовой адресацией для компьютера, которому в основном требуется доступ к 32-разрядным словам. Два младших бита обычно равны 00, но следующие три бита распределяются равномерно. Если задействовать эти три бита в качестве номера модуля памяти, последовательно адресуемые слова оказываются в последовательно расположенных модулях. Система памяти, в которой последовательные слова находятся в разных модулях памяти, называется **расслоенной**. Расслоенная память доводит параллелизм до абсолюта, поскольку большая часть обращений к памяти — это обращения по последовательным адресам. Возможна также разработка неблокирующих сетей, в которых для оптимизации трафика предлагаются несколько путей от каждого процессора к каждому модулю памяти.

NUMA-мультипроцессоры

Количество процессоров в UMA-мультипроцессорах с одной шиной обычно ограничивается несколькими десятками, а для мультипроцессоров с перекрестной или многоступенчатой коммутацией требуется дорогое оборудования, к тому же количество процессоров в них не намного больше. Чтобы объединить в одном мультипроцессоре более 100 процессоров, нужно какое-то иное решение. Ранее предполагалось, что все модули памяти имеют одинаковое время доступа. Если не замыкаться на этой концепции, можно прийти к мультипроцессорам **с неодно-**

родным доступом к памяти (NonUniform Memory Access, **NUMA**). Как и UMA-мультипроцессоры, они предоставляют единое адресное пространство для всех процессоров, но в отличие от UMA-машин доступ к локальным модулям памяти происходит быстрее, чем к удаленным. Следовательно, все UMA-программы смогут без изменений работать на NUMA-машинах, но производительность будет хуже, чем на UMA-машине с той же тактовой частотой.

NUMA-машины имеют три ключевые характеристики, которые в совокупности отличают их от других мультипроцессоров:

- существует единое адресное пространство, видимое всеми процессорами;
- доступ к удаленной памяти производится командами `LOAD` и `STORE`;
- доступ к удаленной памяти выполняется медленнее, чем доступ к локальной.

Если время доступа к удаленной памяти не замаскировано кэшированием (кэш отсутствует), такая система называется **NC-NUMA** (No Caching NUMA — NUMA без кэширования). Если присутствуют согласованные кэши, то система называется **CC-NUMA** (Coherent Cache NUMA — NUMA с согласованными кэшами). Программисты часто называют такую систему **аппаратной распределенной общей памятью**, поскольку она, по сути, аналогична распределенной общей памяти (DSM), реализованной программно, однако поддерживается аппаратно с использованием страниц маленького размера.

Одной из первых NC-NUMA-машин (хотя сам термин в то время еще не существовал) был мультипроцессор Cm* производства Carnegie-Mellon. Он упрощенно показан на рис. 8.28 [Swan et al., 1977]. Этот мультипроцессор состоял из набора процессоров LSI-11, каждый с собственной памятью, обращение к которой производилось по локальной шине. (LSI-11 — однопроцессорная версия очень популярного в 70-е годы мини-компьютера DEC PDP-11.) Кроме того, процессоры LSI-11 были связаны друг с другом системной шиной. Когда выполнялось обращение к памяти, запрос попадал к диспетчеру памяти, который проверял, находится нужное слово в локальной памяти или нет. Если да, запрос направлялся по локальной шине, если нет, запрос направлялся по системной шине к той системе, которая содержала данное слово. Естественно, вторая операция требовала гораздо больше времени, чем первая. Выполнение программы, хранящейся в удаленной памяти, занимало в 10 раз больше времени, чем выполнение той же программы, расположенной локально.

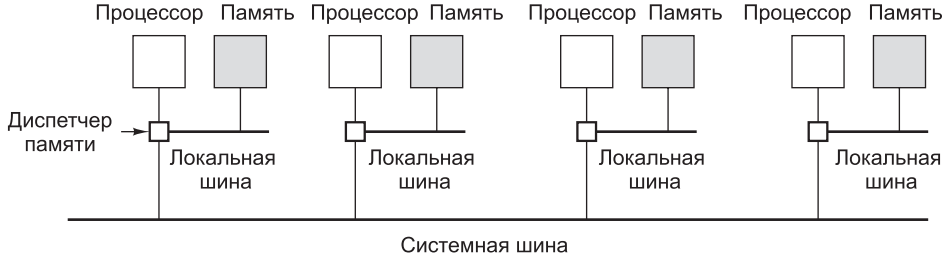


Рис. 8.28. NUMA-машина с двумя уровнями шин

Согласованность памяти в NC-NUMA-машине гарантирована, поскольку в ней отсутствует кэш-память. Каждое слово памяти может находиться только

в одном месте, поэтому нет никакой опасности появления копии с устаревшими данными — здесь вообще нет копий. То, в каком именно модуле памяти находится та или иная страница, имеет большое значение, поскольку от этого зависит производительность. Для максимального увеличения производительности в NC-NUMA-машинах была реализована следующая сложная схема программной поддержки перемещения страниц.

Обычно каждые несколько секунд запускался специальный «сторожевой» процесс (демон), называемый **страничным сканером**. Его задача — следить за статистикой использования страниц и перемещать их таким образом, чтобы росла производительность. Если страница оказывалась в «неправильном» месте, страничный сканер выгружал ее из памяти, поэтому следующее обращение к ней вызывало ошибку отсутствия страницы. Когда происходит ошибка отсутствия страницы, принимается решение о том, куда эту страницу поместить (возможно, не в тот модуль памяти, в котором она была раньше). Для предотвращения пробуксовки существовало правило, согласно которому страница после размещения должна оставаться на месте в течение времени ΔT. Предлагалось множество других алгоритмов, но ни один из них так и не стал безусловно лучшим [LaRowe and Ellis, 1991]. Оптимальная производительность зависит от приложения.

CC-NUMA-мультипроцессоры

Мультипроцессоры, подобные тому, который изображен на рис. 8.28, плохо поддаются масштабированию, поскольку в них нет кэш-памяти. Каждый раз обращаться к удаленной памяти, чтобы получить доступ к слову, которого нет в локальной памяти, очень расточительно — это весьма негативно сказывается на производительности. Однако с добавлением кэш-памяти возникает проблема согласованности кэшей. Один из способов поддержания согласованности кэшей — обеспечить слежение за системной шиной. Технически это сделать несложно, но когда количество процессоров переваливает за некоторый предел, подобное решение становится нереализуемым. Для создания действительно больших мультипроцессоров нужен совершенно другой подход.

Самый популярный на сегодня подход к построению больших мультипроцессоров, относящихся к системам **CC-NUMA**, реализован в **мультипроцессоре на основе каталога**. Основная идея состоит в хранении базы данных с информацией о том, где именно находится каждая строка кэша и каково ее состояние. При обращении к строке кэша в базу данных направляется запрос о том, где эта строка находится и является она «чистой» или «грязной» (модифицированной). Поскольку запрашивать базу данных приходится при выполнении любой команды обращения к памяти, база данных должна поддерживаться высокоскоростным специализированным аппаратным обеспечением, способным обработать запрос за доли шинного цикла.

Чтобы лучше понять, что собой представляет мультипроцессор на основе каталога, рассмотрим в качестве примера систему из 256 узлов, в которой каждый узел состоит из одного процессора и 16-мегабайтного ОЗУ, связанного с процессором локальной шиной. Общий объем памяти составляет 232 байт. Она разделена на 226 строк кэша по 64 байт каждая. Память статически распределена по узлам: адреса 0–16 М располагаются в узле 0, адреса 16–32 М — в узле 1 и т. д. Узлы связаны коммуникационной сетью (рис. 8.29, *а*). Сеть может быть

реализована в виде решетки, гиперкуба или иметь другую топологию. Каждый узел содержит элементы каталога для 218 64-байтных строк кэша, образующих 224 байт памяти. На данный момент мы предполагаем, что строка может содержаться не более чем в одном кэше.

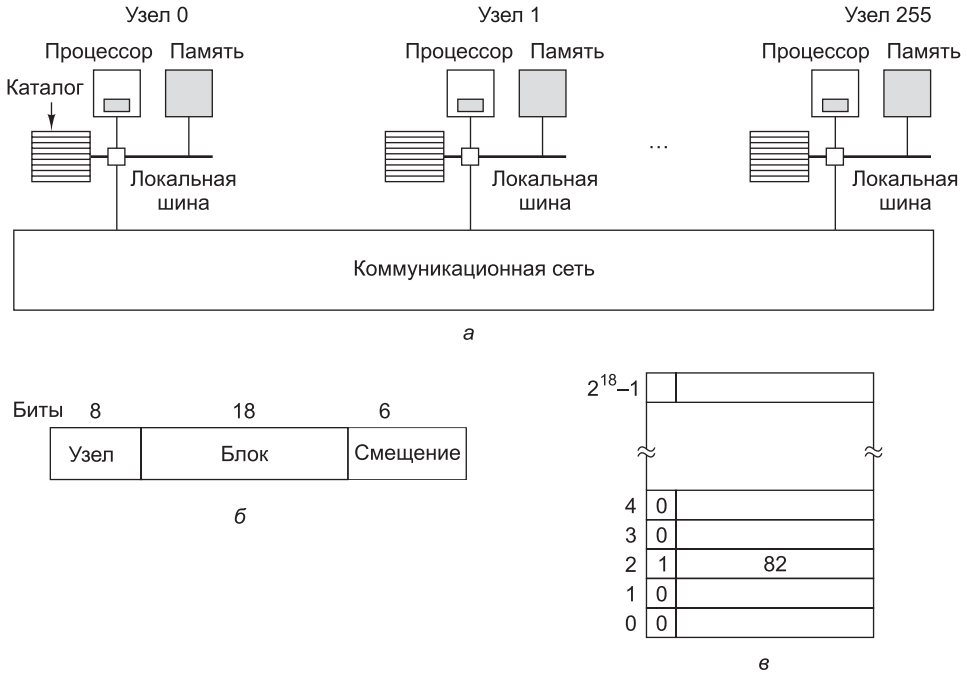


Рис. 8.29. Мультипроцессор на основе каталога, содержащий 256 узлов (*а*); разбиение 32-разрядного адреса памяти на поля (*б*); каталог в узле 36 (*в*)

Чтобы понять, как работает каталог, проследим путь команды `LOAD` из процессора 20, который обращается к кэшированной строке. Сначала процессор, выдавший команду, передает ее диспетчеру памяти, который транслирует ее, чтобы получить физический адрес, например 0x24000108. Диспетчер памяти разделяет этот адрес на три части, как показано на рис. 8.29, *б*. В десятичной системе счисления эти три части представляют собой узел 36, строку 4 и смещение 8. Диспетчер памяти видит, что слово памяти, к которому производится обращение, находится в узле 36, а не в узле 20, поэтому посылает запрос через сеть в узел 36, где находится нужная строка, узнает, есть ли строка 4 в кэше, и если да, то где именно.

Когда запрос приходит в узел 36, он направляется в устройство каталога. Устройство проверяет таблицу из 218 элементов (один элемент на каждую строку кэша) и извлекает элемент 4. На рис. 8.29, *в* видно, что строка отсутствует в кэше, поэтому устройство вызывает строку 4 из локального ОЗУ, отправляет ее узлу 20 и обновляет элемент каталога 4, показывая, что эта строка находится в кэше узла 20.

А теперь рассмотрим второй запрос, на этот раз для строки 2 из узла 36. На рис. 8.29, *в* видно, что эта строка находится в кэше узла 82. В этот момент устройство может обновить элемент 2 каталога, показывая, что строка находится

теперь в узле 20, затем послать сообщение в узел 82, чтобы строка из него была передана в узел 20, и объявить кэш узла 82 недействительным. Отметим, что передавать многочисленные сообщения приходится в любом мультипроцессоре с общей памятью.

Давайте вычислим, сколько памяти занимают каталоги. Каждый узел содержит 16-мегабайтное ОЗУ и 218 9-разрядных элементов для трассировки этого ОЗУ. Таким образом каталог отнимает примерно 9×218 бит от 16 Мбайт, или около 1,76 %, что вполне допустимо. Даже если длина строки кэша составляет 32 байта, потери памяти составят всего 4 %. Если длина строки кэша равна 128 байт, потери окажутся еще ниже — 1 %.

Очевидным недостатком этой схемы является то, что строка может быть кэширована только одним узлом. Чтобы строки можно было кэшировать в нескольких узлах, требуется какой-то способ их поиска (например, чтобы объявлять недействительными или обновлять при записи). Возможны различные варианты.

Один из вариантов — предоставить каждому элементу каталога k полей для идентификации других узлов, что позволит кэшировать каждую строку на нескольких узлах (до k узлов). Второй вариант — заменить поле номера узла битовой картой, по одному биту на узел. Здесь нет ограничений на количество копий, но существенно растут непроизводительные затраты. Каталог, содержащий 256 бит для каждой 64-байтной (512-разрядной) строки кэша, требует более 50 % памяти. Третий вариант — хранить в каждом элементе каталога 8-разрядное поле и использовать это поле в качестве заголовка связного списка, объединяющего вместе все копии строки кэша. При такой стратегии требуется дополнительное пространство на каждом узле для указателей связного списка. Кроме того, требуется просматривать список, чтобы в случае необходимости найти все копии. Каждая из трех стратегий имеет свои достоинства и недостатки, и все три стратегии применяются на практике.

У данной схемы есть еще одна проблема — как проверять, обновлена исходная память или нет? Если нужно считать строку кэша, которая не изменялась, запрос может быть удовлетворен из основной памяти, и при этом не нужно направлять запрос в кэш. Если же требуется считать измененную строку кэша, запрос должен быть направлен в тот узел, в котором находится эта строка кэша, поскольку только здесь имеется ее действительная копия. Если же разрешается иметь только одну копию строки кэша, как показано на рис. 8.29, то контролировать изменения в строках кэша нет никакого смысла, поскольку любой новый запрос требует передачи сообщения существующей копии, чтобы объявить ее недействительной.

Естественно, для хранения информации о состоянии каждой строки кэша («чистая» или «грязная») требуется информировать исходный узел о каждом изменении строки кэша, даже если существует только одна копия этой строки. Если же таких копий несколько, изменение одной из них означает необходимость объявления всех остальных недействительными, поэтому нужен какой-то протокол, позволяющий избежать ситуации гонок. Например, чтобы изменить совместно используемую строку кэша, один из держателей этой строки *перед* ее изменением мог бы запросить монопольный доступ к ней. Такой запрос означал бы объявление всех остальных копий недействительными. Другие возможные оптимизации CC-NUMA-машин обсуждаются в [Cheng and Carter, 2008].

NUMA-мультипроцессор Sun Fire E25K

В качестве примера NUMA-машин с общей памятью рассмотрим семейство мультипроцессоров Sun Fire компании Sun Microsystems. В этом семействе имеется много различных моделей, из которых мы рассмотрим мультипроцессор E25K, содержащий 74 процессора UltraSPARC IV. По своей сути, каждый из этих процессоров представляет собой пару процессоров UltraSPARC III Cu с общими кэшем и памятью. Система E15K отличается только тем, что вместо сдвоенных в ней используются одиночные процессоры. В семействе есть и более простые модели, но нам интересно выяснить, как работают модели с максимальным числом процессоров.

Система E25K содержит 18 наборов плат, каждый набор состоит из платы процессор-память, платы ввода-вывода с четырьмя PCI-слотами и платы расширения. Плата расширения попарно объединяет платы процессор-память и ввода-вывода, связывая эти пары с центральной панелью, которая несет остальные платы и обеспечивает их коммутацию. На каждой плате процессор-память находится 4 процессора и 4 модуля ОЗУ по 8 Гбайт. Таким образом, на каждой плате процессор-память имеется 8 процессоров и 32-гигабайтное ОЗУ (для E15K — 4 процессора и столько же 32-гигабайтных ОЗУ). В целом в системе E25K имеются 144 процессора, 576 Гбайт памяти и 72 PCI-слота, как показано на рис. 8.30. Любопытно, что число 18 было выбрано исключительно из соображений габаритов: система из 18 наборов плат — это самая большая система, которую можно внести в дверной проем, не разбирая на части. В то время как программисты думают только о нулях и единицах, разработчикам приходится задумываться, в том числе и о том, как покупатель будет вносить их творение в дом.

Для соединения 18 наборов плат на центральной панели имеются три схемы перекрестной коммутации размером 18×18 — по одной схеме для адресных линий, ответов и данных. В дополнение к 18 наборам плат, в центральную панель вставляется плата управления системой, содержащая, помимо процессора, интерфейсы для подключения дисковода, ленточного накопителя, последовательных линий и прочей периферии, необходимой для загрузки системы, ее поддержки и управления.

Подсистема памяти — сердце любого мультипроцессора. Как же распределенная память соединяется со 144 процессорами? Прямолинейные подходы, то есть большая общая шина слежения или схема перекрестной коммутации размером 144×72, в данном случае малопригодны. Первый вариант плох тем, что общая шина становится узким местом всей системы, а второй не годится потому, что разрабатывать такой коммутатор сложно и дорого. Поэтому большим мультипроцессорам, таким как E25K, неизбежно приходится использовать нетривиальные подсистемы памяти.

На уровне наборов плат логика слежения обеспечивает каждому процессору возможность сверять поступающие запросы со списком блоков в его локальном кэше. Когда процессор обращается к слову памяти, он сначала преобразует виртуальный адрес в физический и проверяет, присутствует ли нужный блок в кэше. (Физические адреса 43-разрядные, но из-за габаритных требований объем памяти ограничен значением 576 Гбайт.) Если нужный блок обнаруживается в собственном кэше, затребованное слово возвращается. В противном случае логика слежения проверяет, есть ли нужный блок в пределах того же набора плат.

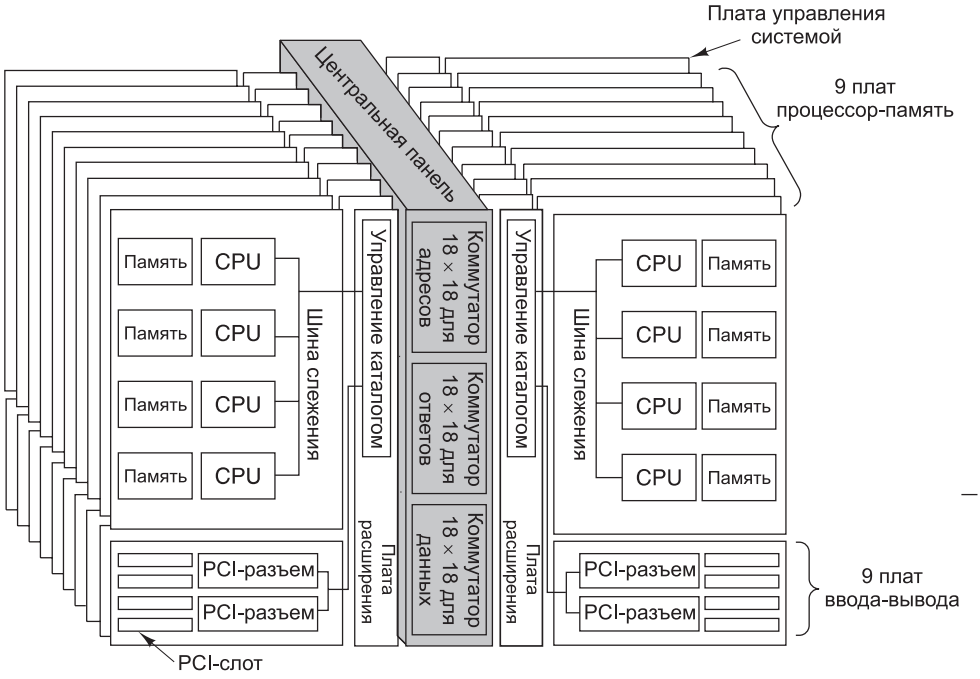


Рис. 8.30. Мультипроцессор E25K компании Sun Microsystems

Если есть, то запрос выполняется. Иначе, как показано далее, запрос посылается через схему перекрестной коммутации адресных линий. Логика слежения способна обслуживать по одному запросу за такт. Так как тактовая частота системы составляет 150 МГц, можно обработать 150 млн запросов в секунду, или 2,7 млрд запросов для всех 18 наборов плат.

Хотя на рис. 8.30 логика слежения показана в виде шины, физически она представляет собой древовидную структуру устройств, вверх и вниз по которой передаются команды. Когда с шины PCI или процессора поступает адрес, он через двухточечное соединение попадает в адресный повторитель, как показано на рис. 8.31. С любого из двух повторителей адрес попадает в плату расширения, откуда передается обратно вниз по дереву к устройствам. Благодаря этой схеме удается избавиться от шины, соединяющей три платы.

Для обмена данными применяется четырехуровневое соединение (рис. 8.29), обеспечивающее высокую производительность. На уровне 0 пары процессоров и блоков памяти соединяются небольшими схемами перекрестной коммутации, которые к тому же соединяются с уровнем 1. Две группы пар процессор-память соединяются второй схемой перекрестной коммутации на уровне 1. Эти коммутаторы выполнены в виде специализированных интегральных схем. У них есть входы для каждой строки и каждого столбца коммутирующей сетки, хотя не все сочетания строк и столбцов используются (или даже имеют смысл). Вся логика коммутации плат построена на схемах перекрестной коммутации размером 3×3.

Каждый набор плат состоит из трех плат: платы процессор-память, платы ввода-вывода и платы расширения, соединяющей две предыдущие платы.

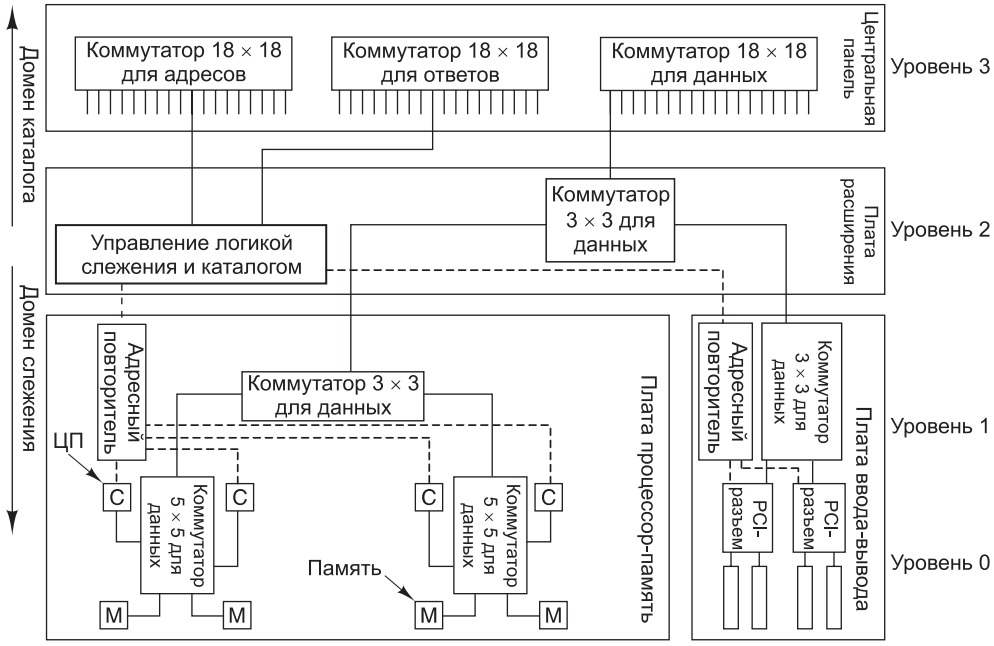


Рис. 8.31. Четырехуровневое соединение блоков в Sun Fire E25K. Пунктирные линии означают передачу адресов, сплошные — передачу данных

Коммутатор уровня 2 расположен на плате расширения, он соединяет саму память и порты ввода-вывода (которые во всех процессорах UltraSPARC отображаются на память). Все данные, поступающие в набор плат или из него, проходят через коммутатор уровня 2. Наконец, обмен данными между разными платами на уровне 3 происходит через схему перекрестной коммутации размером 18 × 18. Данные передаются блоками по 32 байта, таким образом, на передачу стандартного блока в 64 байта требуется два такта.

Теперь, выяснив, как расположены компоненты, обратимся к общей памяти. На самом нижнем уровне адресное пространство объемом 576 Гбайт разбивается на 229 блоков по 64 байта. Это — неделимые элементы памяти. У каждого из них есть своя «родная» плата, где блок «живет», пока он не потребуется где-то еще. Большинство блоков большую часть времени находятся на своих платах. Когда процессору требуется блок, будь то с собственной платы или с любой другой из 17 оставшихся, он сначала запрашивает копию в собственном кэше, после чего работает с кэшированной копией. Хотя на каждой микросхеме в системе E25K находятся два процессора, у них общее адресное пространство, а значит, и общий кэш блоков.

Каждый блок памяти (и каждая строка кэша всех микросхем) может находиться в одном из трех состояний:

- эксклюзивный доступ (для записи);
- совместный доступ (для чтения);
- недействителен (то есть пуст).

Когда процессору требуется записать в память слово или считать его из памяти, он, прежде всего, проверяет собственный кэш. Если слово там не обнаруживается, инициируется локальный запрос физического адреса, который широковещательно распространяется в пределах своего набора плат. Если нужный блок обнаруживается в кэше набора плат, логика слежения определяет факт кэш-попадания и отвечает на запрос. Если строка находится в эксклюзивном доступе, она передается запросившему ее процессору, а исходная копия помечается как недействительная. Если же строка находится в совместном доступе, она не передается запросившему ее процессору, так как память сама отправляет ответ, когда очищается строка кэша.

Если логике слежения не удается найти строку кэша или найденная строка находится в совместном доступе, через центральную панель к родной плате искомого блока передается запрос, чтобы узнать, где находится блок. Состояние каждого из блоков хранится в его ECC-битах, поэтому плата может немедленно выяснить это состояние. Если блок не находится в совместном доступе или находится в совместном доступе одной или нескольких удаленных плат, память на родной плате будет обновлена, поэтому родная плата сможет выполнить запрос. В этом случае копия строки кэша за два такта передается через схему перекрестной коммутации данных запросившему ее процессору.

Если делается запрос на чтение, в каталог родной платы блока вносится информация о том, что данную строку кэша использует еще один клиент (то есть она находится в совместном доступе), и на этом транзакция заканчивается. Если же делается запрос на запись, всем платам, имеющим копию блока (если такие есть), передается сообщение о том, что блок более недействителен. Благодаря этому после запроса на запись остается только одна копия блока.

Предположим теперь, что блок находится в эксклюзивном доступе удаленной платы. Когда родная плата получает запрос, она по каталогу находит адрес нужной удаленной платы и отвечает на запрос сообщением с информацией о том, где находится строка кэша. После этого отправитель посылает новое сообщение найденной плате. Когда та получает запрос, она отправляет в ответ требуемую строку кэша. После этого, в случае запроса на чтение, строка помечается как находящаяся в совместном доступе, и ее копия отсылается на родную плату. В случае же запроса на запись отвечающая сторона объявляет свою копию недействительной, тем самым предоставляя отправителю запроса эксклюзивную копию.

Так как каждая плата содержит 229 блоков памяти, в худшем случае каталог должен иметь 229 записей. Поскольку в действительности его объем гораздо меньше, может оказаться, что в каталоге (поиск в котором осуществляется ассоциативно) нет места для некоторых записей. В этом случае родному набору плат придется посылать широковещательный запрос всем остальным 17 наборам плат, чтобы определить местоположение блока. Обязанности по поддержанию согласованности каталогов и выполнению протокола обновления возлагаются на схему перекрестной коммутации ответов, которая обрабатывает большую часть трафика, направленного обратно отправителю. Благодаря разделению протокольного трафика по двум шинам (адресов и ответов) и передаче данных по третьей, общую пропускную способность системы удается поддерживать на высоком уровне.

За счет распределения нагрузки между разными устройствами на разных платах Sun Fire E25K может работать с очень высокой производительностью.

Ранее уже упоминалось значение 2,7 млрд запросов в секунду. Центральная панель способна поддерживать девять одновременных обменов данными с девятью платами-отправителями и девятью платами-получателями. Так как схема перекрестной коммутации данных имеет ширину 32 байта, за каждый такт может передаваться 288 байт данных. На тактовой частоте 150 МГц это дает пиковую пропускную способность 40 Гбайт/с, когда все обращения направлены удаленным платам. Если же есть возможность программно расположить страницы памяти так, чтобы большая часть обращений были бы локальными, пиковая пропускная способность системы будет заметно превышать это значение.

Более подробную техническую информацию о Sun Fire E25K смотрите в [Charlesworth, 2002; Charlesworth, 2001].

В 2009 году компания Oracle приобрела Sun Microsystems, и разработка серверов на базе SPARC была продолжена. Модель SPARC Enterprise M9000 является наследником E25K. M9000 содержит быстрые четырехъядерные процессоры SPARC, а также дополнительную память и слоты PCI. Полностью укомплектованный сервер M9000 содержит 256 процессоров SPARC, 4 Тбайт DRAM и 128 интерфейсов ввода-вывода PCIe.

COMA-мультипроцессоры

NUMA- и CC-NUMA-машины обладают одним серьезным недостатком: обращения к удаленной памяти выполняются гораздо медленнее, чем к локальной. В CC-NUMA-машине эта разница в производительности в какой-то степени нивелируется за счет кэш-памяти. Однако если объем запрашиваемых удаленных данных значительно превышает вместимость кэш-памяти, постоянно будут происходить кэш-промахи, что негативно скажется на производительности.

Мы уже знаем, что достаточно высокую производительность имеют UMA-машины, но число процессоров в них невелико, к тому же они довольно дороги. NC-NUMA-машины хорошо масштабируются, но в них требуется ручное или полуавтоматическое размещение страниц памяти, результаты которого часто плачевны. Дело в том, что очень непросто предсказать, где и какие страницы могут понадобиться, кроме того, страницы трудно перемещать из-за их больших размеров. CC-NUMA-машины, такие как мультипроцессор Sun Fire E25K, начинают работать очень медленно, если большому числу процессоров требуются большие объемы удаленных данных. Так или иначе, каждая из этих схем имеет существенные недостатки.

Однако существует мультипроцессор, в котором все эти проблемы решаются за счет использования основной памяти каждого процессора в качестве кэш-памяти. Такая система называется **COMA (**Cache Only Memory Access — **доступ только к кэш-памяти)**. В ней страницы не имеют собственных «домашних» машин, как в системах NUMA и CC-NUMA, фактически, страницы в этой системе вообще не имеют «прописки».

Вместо этого физическое адресное пространство делится на строки кэша, которые по запросу свободно перемещаются в системе. Блоки памяти не имеют собственных машин. У них, как у кочевников в некоторых странах третьего мира, дом там, где они оказались. Память, которая привлекает строки по мере необходимости, называется **притягивающей**. Использование основной памяти

в качестве большого кэша увеличивает процент кэш-попаданий, а следовательно, и производительность.

К сожалению, ничего идеального не бывает. С системой COMA связаны две новые проблемы:

- Как размещаются строки кэша?
- Что делать, когда удаляемая из памяти строка является последней копией?

Первая проблема связана со следующим фактом. Как известно, диспетчер памяти выполняет трансляцию виртуального адреса в физический. Если после трансляции оказывается, что строки нет в «настоящем» аппаратном кэше, очень трудно сказать, есть вообще искомая строка в основной памяти или ее там нет. Аппаратная поддержка механизма разбиения памяти на страницы здесь не поможет, поскольку каждая страница состоит из большого количества отдельных строк кэша, которые располагаются в системе независимо друг от друга. Даже если известно, что строки в основной памяти нет, как выяснить, где она есть? В данном случае нельзя спросить об этом «домашнюю» машину потерявшейся страницы, поскольку таковой машины в системе просто нет.

Было предложено несколько решений этой проблемы. Чтобы знать, находится строка кэша в основной памяти или нет, для каждой строки кэша можно аппаратно поддерживать специальный тег. Тогда диспетчер памяти сможет сравнивать тег нужной строки с тегами всех строк кэша, пока не обнаружится совпадение.

Другое решение — отображать страницы целиком, но при этом не требовать наличия всех строк кэша. Тогда для каждой страницы потребуется аппаратно построить битовую карту, где каждой строке соответствует один бит, который и укажет на присутствие или отсутствие этой строки. В этой схеме, которая называется **простой схемой COMA**, если строка присутствует, она должна находиться в правильной позиции на своей странице. Если она отсутствует, то любая попытка использовать ее должна вызывать исключение, которое позволит программно найти и задействовать нужную строку.

Таким образом, система будет искать только те строки, которые действительно находятся в удаленной памяти. Еще одно решение — предоставить каждой странице «домашнюю» машину (домашнюю в том смысле, что в каталоге для нее выделяется запись, а не в том, что на этой машине хранятся данные). Тогда, чтобы выяснить, где искать строку, можно отправить сообщение ее домашней машине. Другое решение — организовать память в виде древовидной структуры и искать, двигаясь вверх, пока строка не будет найдена.

Вторая проблема связана с удалением последней копии. Как и в CC-NUMA-машине, строка кэша может одновременно находиться в нескольких узлах. Если происходит кэш-промах, строку нужно прочитать, а это обычно означает ее удаление. А что произойдет, если выбранная строка окажется последней копией? В этом случае ее нельзя удалять.

Одно из возможных решений — вернуться к каталогу и проверить, существуют ли другие копии. Если да, то строку можно смело удалять. Если нет, ее нужно где-то разместить. Другое решение — пометить одну из копий каждой строки кэша как главную и никогда ее не удалять. При таком подходе проверять каталог не потребуется. В любом случае COMA-машина потенциально должна иметь более высокую производительность, чем CC-NUMA, но пока было создано

всего несколько COMA-машин, а для реализации всего их потенциала нужно накопить некоторый опыт. Первыми COMA-машинами были KSR-1 [Burkhardt et al., 1992] и Data Diffusion Machine [Hagersten et al., 1992]. Более современные публикации о COMA — [Vu et al., 2008; Zhang and Jesshope, 2008].

Мультикомпьютеры

Как показано на рис. 8.20, в категорию MIMD входят два вида процессоров с параллельной архитектурой: мультипроцессоры и мультикомпьютеры. В предыдущем разделе мы рассматривали мультипроцессоры. Мы выяснили, что мультипроцессоры могут иметь общую память, доступ к которой выполняется обычными командами `LOAD` и `STORE`. Для реализации такой памяти может использоваться множество схем, включая шины слежения, сети с перекрестной и многоступенчатой коммутацией, различные схемы на основе каталога. Во всех случаях программы, написанные для мультипроцессора, могут получать доступ к любому месту в памяти, не имея никакой информации о внутренней топологии или схеме реализации. Именно благодаря такой иллюзии мультипроцессоры весьма популярны у пользователей и программистов.

Однако мультипроцессорам свойственны и некоторые недостатки, и это автоматически означает усиление роли мультикомпьютеров. В первую очередь, мультипроцессоры плохо масштабируются. Мы уже знаем, сколько разнообразных устройств потребовалось инженерам компании Sun ввести в систему E25K, чтобы поддерживать работу 72 процессоров. Что касается мультикомпьютеров, то далее мы рассмотрим систему из 65 536 процессоров. Пройдут годы, прежде чем кто-нибудь сумеет построить коммерческий мультипроцессор с 65 536 узлами, да и к тому времени в ходу уже будут мультикомпьютеры с миллионами процессоров.

Далее, на производительности мультипроцессора может серьезно сказываться конкуренция за доступ к памяти. Если 100 процессоров постоянно пытаются считывать и записывать одни и те же переменные, конкуренция за ресурсы модулей памяти, шин и каталогов может сильно ударить по производительности.

Вследствие этих и других факторов разработчики проявляют повышенный интерес к таким параллельным компьютерным архитектурам, в которых каждый процессор имеет собственную память, недоступную напрямую для других процессоров. Это — мультикомпьютеры. Поскольку программы на разных процессорах в мультикомпьютере не могут получить доступ к памяти других процессоров командами `LOAD` и `STORE`, они взаимодействуют друг с другом с помощью примитивов `send` и `receive`, которые используются для передачи сообщений. Это отличие полностью меняет модель программирования.

Каждый узел в мультикомпьютере состоит из одного или нескольких процессоров, ОЗУ (общего для процессоров только данного узла), дисковода и (или) других устройств ввода-вывода, а также коммуникационного процессора. Коммуникационные процессоры связаны между собой высокоскоростной коммуникационной сетью (см. подраздел «UMA-мультипроцессоры в симметричных мультипроцессорных архитектурах»). Используется множество различных топологий, схем коммутации и алгоритмов выбора маршрута, однако у всех

мультикомпьютеров есть общая черта: когда программа выполняет примитив send, коммуникационный процессор извещается об этом и передает блок данных в целевую машину (возможно, после предварительного запроса и получения разрешения). Обобщенная схема мультикомпьютера показана на рис. 8.32.

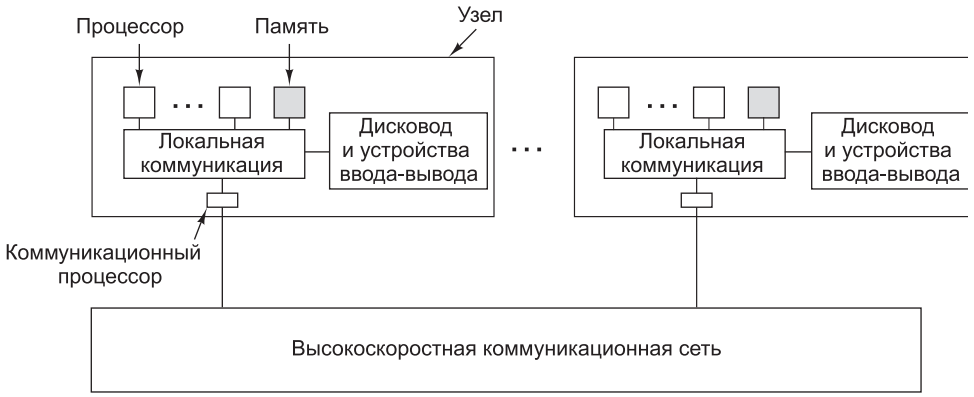


Рис. 8.32. Схема мультикомпьютера

Коммуникационные сети

Как показано на рис. 8.32, мультикомпьютеры связываются друг с другом через коммуникационные сети. Рассмотрим их подробнее. Интересно отметить, что мультикомпьютеры и мультипроцессоры в этом отношении очень похожи, поскольку мультипроцессоры часто содержат несколько модулей памяти, которые также должны связываться друг с другом и с процессорами. Следовательно, многое из того, о чем мы будем говорить в этом подразделе, применимо к обоим типам параллельных компьютерных архитектур.

Основная причина сходства коммуникационных связей в мультипроцессоре и мультикомпьютере заключается в том, что в обоих случаях имеет место передача сообщений. Даже в однопроцессорной машине, когда процессору нужно считать или записать слово, он активизирует определенные линии на шине и ждет ответа. Это примерно то же самое, что и обмен сообщениями: инициатор посылает запрос и ждет ответа. В больших мультипроцессорах при взаимодействии между процессорами и удаленной памятью процессор почти всегда посылает в память сообщение, так называемый **пакет**, в котором запрашиваются те или иные данные, а память посылает процессору ответный пакет.

Топология

Топология коммуникационной сети определяет схему размещения линий связи и коммутаторов (это может быть, например, кольцо или решетка). Топологию сетей принято изображать в виде графов, в которых дуги соответствуют линиям связи, а узлы — коммутаторам (рис. 8.33). С каждым узлом в сети (или в соответствующем графе) связан определенный набор линий связи. Математики называют число линий **степенью** узла, инженеры — **коэффициентом разветвления**. Чем больше степень, тем больше вариантов маршрута и тем выше отказоустойчивость. Если каждый узел содержит k дуг и связи выполнены правильно, можно

построить коммуникационную сеть так, чтобы она оставалась полносвязной, даже если повреждены $k - 1$ линий.

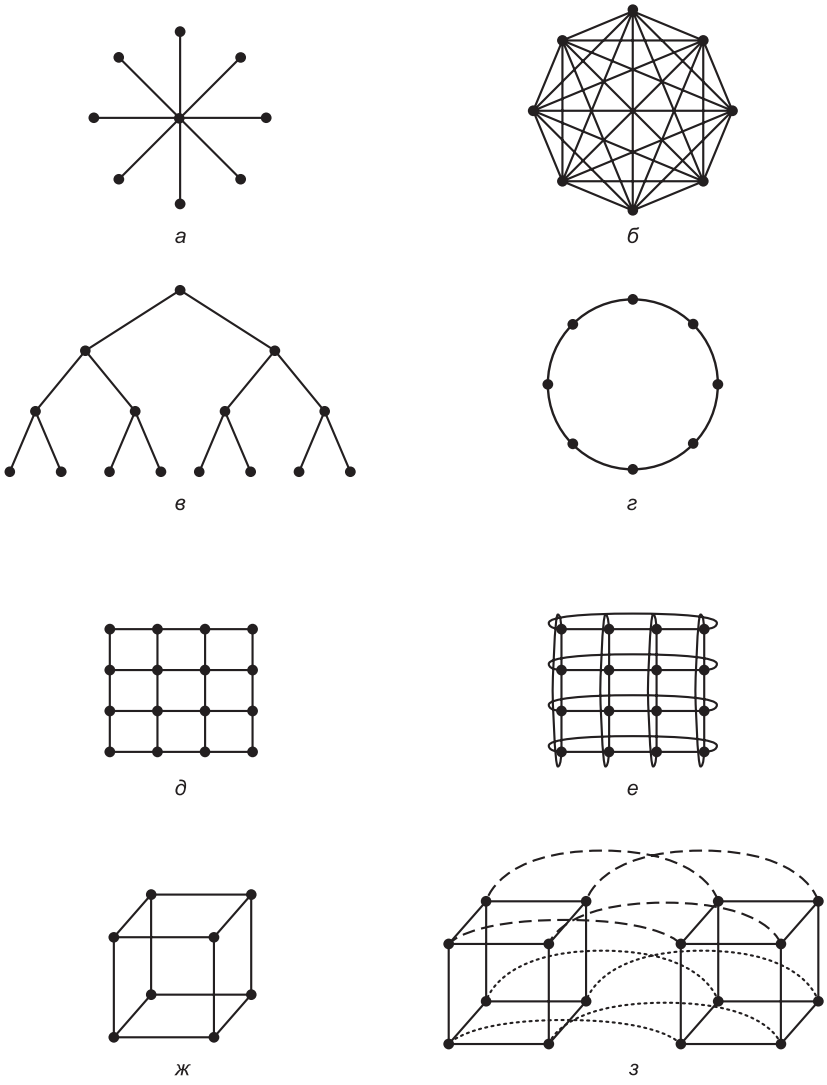


Рис. 8.33. Различные топологии. Жирные точки соответствуют коммутаторам. Процессоры и модули памяти не показаны: звезда (*а*); полная взаимосвязь (*б*); дерево (*в*); кольцо (*г*); решетка (*д*); двойной тор (*е*); куб (*ж*); четырехмерный гиперкуб (*з*)

Следующая характеристика коммуникационной сети — ее **диаметр**. Если расстоянием между двумя узлами мы будем считать число дуг, которые нужно пройти, чтобы попасть из одного узла в другой, то диаметром графа является расстояние между двумя узлами, расположенными дальше всех друг от друга. Диаметр сети определяет самую большую задержку при передаче пакетов от одного процессора к другому или от процессора к памяти, поскольку каждая

пересылка через линию связи занимает определенное время. Чем меньше диаметр, тем выше производительность. Также имеет большое значение среднее расстояние между парой узлов, поскольку от него зависит среднее время передачи пакета.

Еще одна важная характеристика коммуникационной сети — ее пропускная способность, то есть объем данных, которые она способна передавать в секунду. Наиболее полезной метрикой пропускной способности является **пропускная способность сечения**. Чтобы вычислить это значение, нужно мысленно разделить коммуникационную сеть на две равные (с точки зрения числа узлов) несвязанные части путем удаления ряда дуг из графа, а затем посчитать общую пропускную способность удаленных дуг. Пропускная способность сечения — это минимальная (для всех доступных вариантов) пропускная способность. Например, пропускная способность сечения, равная 800 бит/с, означает, что если между двумя частями сети много связей, общая пропускная способность в худшем случае составит только 800 бит/с. По мнению многих разработчиков, пропускная способность сечения — самая важная характеристика коммуникационной сети. Часто основная цель, которую ставят разработчики коммуникационной сети, — добиться максимальной пропускной способности сечения.

Коммуникационные сети можно характеризовать по их **размерности**. Размерность определяется числом возможных вариантов перехода от источника к приемнику. Если выбора нет (то есть существует только один путь от каждого источника к каждому приемнику), то сеть нульмерная. Если есть два возможных варианта (например, либо на восток, либо на запад), то сеть одномерная. Если есть две оси и пакет может направиться на восток или на запад либо на север или на юг, то говорят, что такая сеть двухмерная и т. д.

На рис. 8.33 показано несколько топологий. Здесь изображены только линии связи и коммутаторы (в виде точек). Модули памяти и процессоры (они на рисунке не показаны) соединяются с коммутаторами через интерфейсы. На рис. 8.31, *а* изображена нульмерная конфигурация **звезда**, в которой процессоры и модули памяти подключаются к внешним узлам, а переключение совершает центральный узел. Такая схема очень проста, но в большой системе центральный коммутатор окажется узким местом системы. С точки зрения отказоустойчивости это тоже очень неудачная схема, поскольку отказ одного центрального коммутатора вызывает крах всей системы.

На рис. 8.33, *б* изображена другая нульмерная топология — **полная взаимосвязь**. Здесь каждый узел непосредственно связан со всеми остальными. В такой схеме пропускная способность сечения максимальна, диаметр минимален, а отказоустойчивость очень высока (даже при утрате шести линий связи система все равно остается полносвязной). Однако для k узлов требуется $k(k-1)/2$ каналов, а это совершенно неприемлемо для больших значений k.

Следующая топология — **дерево** (рис. 8.33, *в*). Здесь основная проблема состоит в том, что пропускная способность сечения равна пропускной способности линии связи. Обычно основной трафик наблюдается у верхушки дерева, поэтому верхние узлы становятся узким местом всей системы. Чтобы разрешить эту проблему, нужно увеличить пропускную способность сечения путем увеличения пропускной способности верхних линий связи. Например, самые нижние линии могут иметь пропускную способность b, следующий уровень — $2b$, а каждая ли-

ния верхнего уровня — 4*b*. Такая схема, названная **толстым деревом** (fat tree), применялась в коммерческих мультикомпьютерах Thinking Machines CM-5.

Кольцо (рис. 8.33, *г*) — это одномерная топология, поскольку каждый отправленный пакет может пойти направо или налево. **Решетка**, или **сетка** (рис. 8.33, *д*), — это двухмерная топология, которая применяется во многих коммерческих системах. Она отличается регулярностью и легко масштабируется в сторону увеличения, а ее диаметр составляет квадратный корень от числа узлов (то есть при масщтабировании системы диаметр увеличивается незначительно). **Двойной тор** (рис. 8.33, *е*) является разновидностью решетки, у которой края соединены. Эта топология характеризуется более высокой отказоустойчивостью и меньшим диаметром, чем обычная решетка, поскольку между в ней двумя противоположными узлами всего два хопа.

Еще одна популярная топология — трехмерный тор. Эта топология описывается трехмерной структурой, узлы которой находятся в точках (*i*, *j*, *k*), а все координаты являются целыми в пределах от (1, 1, 1) до (*l*, *m*, *n*). У каждого узла есть шесть соседей, по два вдоль каждой оси координат, а крайние узлы на противоположных краях связаны друг с другом, как и в двухмерных торах.

Куб (рис. 8.33, *ж*) — это регулярная трехмерная топология. На рисунке изображен куб размером $2 \times 2 \times 2$, но в общем случае это может быть куб размером $k \times k \times k$. На рис. 8.31, *з* показан четырехмерный куб, полученный из двух трехмерных кубов, которые связаны между собой. Можно сделать пятимерный куб, соединив вместе 4 четырехмерных куба. Чтобы получить 6 измерений, нужно продублировать блок из четырех кубов и соединить соответствующие узлы и т. д. **Гиперкубом** называется n-мерный куб (рис. 8.33, *з*). Эта топология используется во многих параллельных компьютерных архитектурах, поскольку ее диаметр линейно зависит от размерности. Другими словами, диаметр — это логарифм по основанию 2 от числа узлов, поэтому 10-мерный гиперкуб имеет 1024 узла, но диаметр равен всего 10, что дает очень незначительные задержки при передаче данных. Отметим, что решетка размером 32×32, которая также содержит 1024 узла, имеет диаметр 62, что более чем в шесть раз больше, чем у гиперкуба. Цена, которую приходится платить за меньший диаметр гиперкуба, — увеличение числа разветвлений и, следовательно, линий связи. Тем не менее гиперкуб — основное решение для высокопроизводительных систем.

Мультикомпьютеры имеют столь разнообразные формы и размеры, что выстроить для них сколько-нибудь внятную классификацию очень трудно. Тем не менее можно выделить два основных «стиля» — это процессоры с массовым параллелизмом и кластеры. Мы рассмотрим их по очереди.

Процессоры с массовым параллелизмом

Процессоры с массовым параллелизмом (Massively Parallel Processors, **MPP**) — это огромные суперкомпьютеры стоимостью в несколько миллионов долларов. Они используются в различных отраслях науки и техники для выполнения сложных вычислений, обработки большого числа транзакций в секунду, управления большими базами данных и т. д. Изначально это были суперкомпьютеры, предназначенные в основном для научных расчетов, но сейчас многие из них находят применение в коммерции. В каком-то смысле они являются наслед-

никами мощных мэйнфреймов 1960-х годов (хотя связь между ними столь же эфемерная, как между воробьем и тиранозавром). В целом, можно говорить, что MPP-мультикомпьютеры вытеснили SIMD-машины, векторные суперкомпьютеры и матричные процессоры с вершины «пищевой» компьютерной цепочки.

В большинстве MPP-машин используются стандартные процессоры. Это могут быть процессоры Intel Pentium, Sun UltraSPARC, IBM RS/6000 и DEC Alpha. Отличает мультикомпьютеры наличие высокопроизводительной коммуникационной сети, по которой можно передавать сообщения с низким временем запаздывания и высокой пропускной способностью. Обе характеристики (время запаздывания и пропускная способность) очень важны, поскольку сообщения в основном невелики по размеру (менее 256 байт), хотя при этом главный вклад в общий трафик вносят большие сообщения (более 8 Кбайт). MPP-мультикомпьютеры поставляются вместе с весьма дорогостоящим программным обеспечением и библиотеками.

Еще одна характеристика MPP — огромные объемы ввода-вывода. С помощью MPP-мультикомпьютеры обычно приходится обрабатывать огромные массивы данных, иногда терабайты. Эти данные должны быть распределены по многочисленным дискам, и их с большой скоростью нужно передавать среди устройств машины.

Наконец, важно помнить о еще одной черте MPP — отказоустойчивости. При наличии тысяч процессоров несколько неисправностей в неделю неизбежны. Прекращать работу системы из-за сбоев в одном из процессоров неприемлемо, особенно если ожидается, что сбои будут случаться каждую неделю. Поэтому в больших MPP-машинах всегда имеется специализированная аппаратная и программная поддержка постоянного мониторинга системы, обнаружения неполадок и их исправления.

Теперь, по идее, надо было бы заняться изучением основных принципов организации MPP-машин, но этих принципов, по правде говоря, совсем не много. На данном этапе достаточно знать, что MPP-машина представляет собой ряд более или менее стандартных вычислительных узлов, связанных друг с другом высокоскоростной коммуникационной сетью. Поэтому далее мы просто рассмотрим несколько конкретных примеров MPP-машин, в частности BlueGene/P и Red Storm.

BlueGene

В качестве первого примера процессора с массовым параллелизмом рассмотрим систему IBM BlueGene. Этот проект был задуман IBM в 1999 году как суперкомпьютер для решения вычислительных задач большой сложности в области биологии. В частности, биологи считают, что функции белка определяются его трехмерной структурой. Но определение формы даже одной небольшой молекулы белка на суперкомпьютерах того времени потребовало бы нескольких лет вычислений. При этом в человеческом организме около полумиллиона различных белков, некоторые из них исключительно сложны, и нарушения в структуре любого могут приводить к серьезным наследственным заболеваниям (например, муковисцидозу). Очевидно, что для расчета трехмерной структуры всех человеческих белков требуется на несколько порядков повысить вычислительную мощность, и моделирование формы белковой молекулы — лишь одна из задач,

на решение которых направлен проект BlueGene. Столь же сложные задачи из молекулярной динамики, моделирования климата, астрономии и даже финансового моделирования также требуют совершенствования суперкомпьютеров.

Почувствовав потребность рынка в суперкомпьютерах, в IBM вложили в разработку и постройку BlueGene 100 млн долларов. В ноябре 2001 появился и первый заказчик первого компьютера из семейства BlueGene под названием **BlueGene/L**. Заказчиком стала Ливерморская национальная лаборатория, работающая под началом департамента энергетики США. В 2007 году компания IBM представила компьютер Blue Gene второго поколения **BlueGene/P**, который мы и будем рассматривать.

Целью проекта BlueGene была постройка MPP-машины, которая не только была бы самой быстрой, но и самой эффективной в отношении показателей терафлоп/доллар, терафлоп/ватт и терафлоп/м3. По этой причине в IBM отказались от принципов, которые были положены в основу разработки предыдущих MPP-машин и согласно которым применялись самые быстрые компоненты независимо от их цены. Вместо этого было решено выпустить собственный однокристальный компонент, работающий с умеренной скоростью и обладающий низким энергопотреблением, чтобы на его основе построить большую машину с эффективным расположением компонентов. Первая BlueGene/P была поставлена в немецкий университет в ноябре 2007 года. Система содержала 65 536 микропроцессоров, а ее производительность досгитала 167 терафлоп/с. На момент установки этот суперкомпьютер был самым быстрым в Европе и шестым по быстродействию в мире. Кроме того, BlueGene/P входил в число лидеров по эффективности потребления мощности: он выполнял 371 мегафлоп/Вт, то есть его отдача по мощности была вдвое выше, чем у его предшественника BlueGene/L. Первая модель BlueGene/P в 2009 году была дополнена до 294 912 процессоров, в результате чего ее вычислительная мощь достигла 1 петафлоп/с.

Сердцем системы BlueGene/P является узел, образованный из специализированной микросхемы, структура которой показана на рис. 8.34. Она состоит из четырех ядер PowerPC 450, работающих с частотой 850 МГц. PowerPC 450 — это конвейеризованный сдвоенный суперскалярный процессор, популярный во встраиваемых системах. В каждом ядре имеется пара сдвоенных блоков выполнения операций с плавающей точкой (Floating Point Unit, FPU), что в сумме позволяет за один цикл выполнять 4 команды с плавающей точкой. Эти блоки дополнены поддержкой SIMD-команд, которые могут быть полезны при обработке массивов. Таким образом, в отношении производительности этот процессор никак нельзя причислить к рекордсменам.

На микросхеме поддерживаются три уровня кэширования. Кэш первого уровня раздельный, в нем 32 Кбайта отводится для команд и еще 32 Кбайта — для данных. Размер объединенного кэша второго уровня составляет 2 Кбайта. В действительности, это не столько кэши, сколько буферы предвыборки. В кэшах второго уровня реализован механизм слежения друг за другом, благодаря которому поддерживается их согласованность. Третий уровень представлен объединенным целостным кэшем объемом 4 Мбайта, который совместно используется обоими кэшами второго уровня. Кэши L1 всех четырех процессоров согласованы; таким образом, когда общий блок памяти присутствует сразу в нескольких кэшах, результаты записи в него одним процессором немедленно отражаются

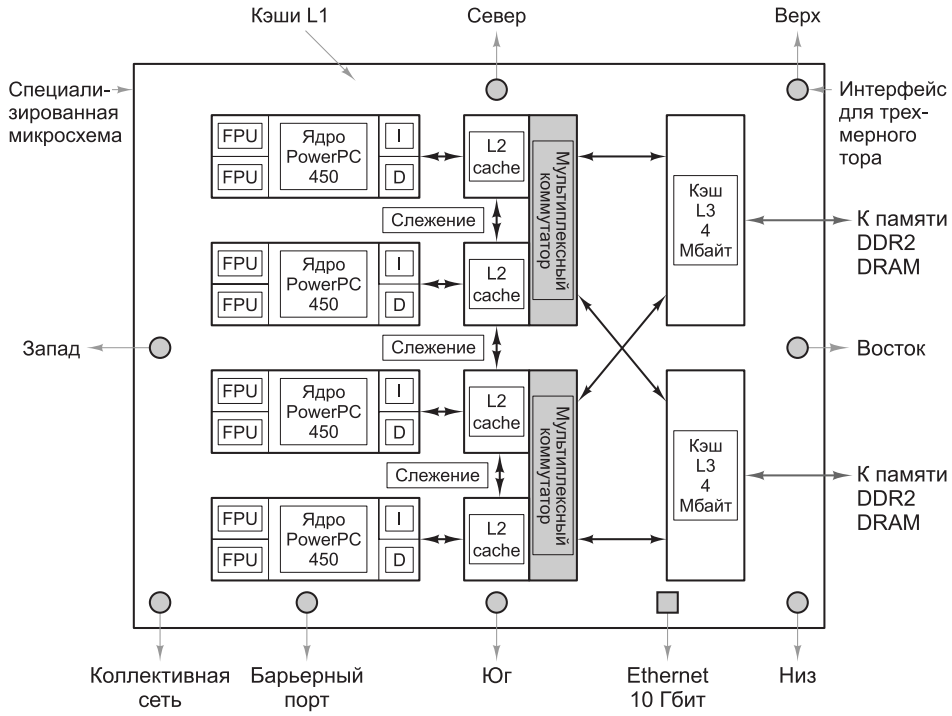


Рис. 8.34. Специализированный микропроцессор в системе BlueGene/P

в кэшах других процессоров. Обращение к памяти, которое вызывает кэш-промах на первом уровне и кэш-попадание на втором, обрабатывается 11 тактов. При кэш-промахе на втором уровне кэш-попадание на третьем обрабатывается уже 28 тактов. Наконец, при кэш-промахе на третьем уровне приходится обращаться к главной памяти (DDR SDRAM), на что требуется около 75 тактов.

Четыре процессора связаны высокопроизводительной шиной в сеть с топологией «трехмерный тор», требующей шести соединений: верх, низ, север, запад, юг и восток. Кроме того, каждый процессор связан через порт с коллективной сетью, используемой для рассылки данных между процессорами. Барьерный порт ускоряет операции синхронизации, предоставляя каждому процессору быстрый доступ к специализированной синхронизирующей сети.

Для более высокого уровня в IBM была разработана специализированная плата, на которую устанавливается одна из микросхем, показанных на рис. 8.35, *а*, а также оперативная память DDR2 на 2 Гбайт. Микросхема показана на 8.35, *а*, плата — на рис. 8.35, *б*.

Платы монтируются на встраиваемой панели, по 32 платы на панель, что дает 32 микросхемы (то есть 128 процессоров) на панель. Так как на каждой плате имеется DRAM объемом 2 Гбайт, всего на панели получается 64 Гбайт памяти (рис. 8.35, *в*). На следующем уровне 32 такие панели вставляются в стойку, в результате в стойке оказываются 4096 процессоров. Стойка показана на рис. 8.35, *г*.

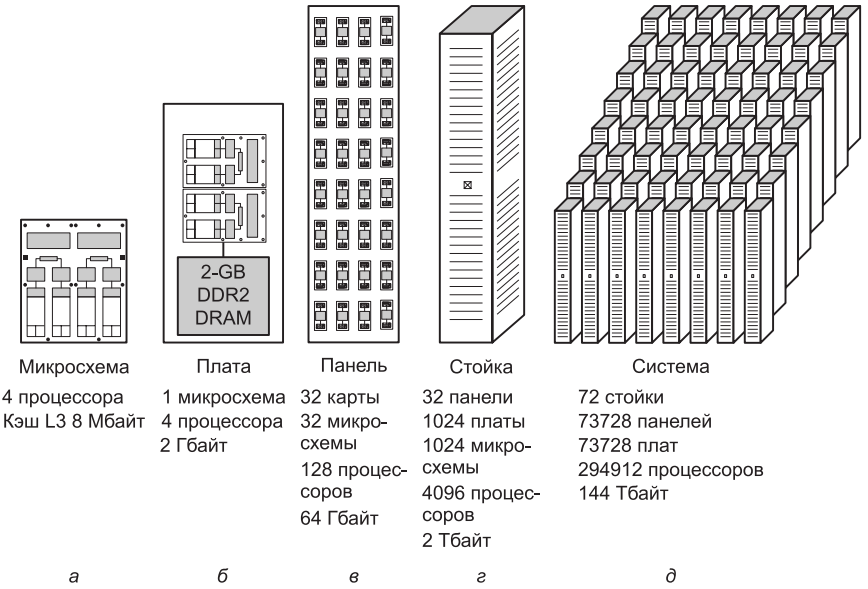


Рис. 8.35. BlueGene/P: микросхема (*а*), плата (*б*), панель (*в*), стойка (*г*), система (*д*)

Наконец, вся система, состоящая из 72 стоек с 294 912 процессоров, изображена на рис. 8.33, *д*. PowerPC 450 обрабатывает до 6 команд/цикл; таким образом, полная система BlueGene/P теоретически может обрабатывать за цикл до 1 769 472 команд. При частоте 850 МГц теоретическая производительность системы достигает 1,504 петафлоп/с. Однако из-за конфликтов данных, задержек памяти и недостаточного параллелизма фактическая производительность существенно снижается. При выполнении реальных программ на BlueGene/P достигалась производительность около 1 петафлоп/с.

Система представляет собой мультикомпьютер в том смысле, что ни один из процессоров не имеет непосредственного доступа к памяти, если не считать 2 Гбайт собственной памяти на плате. Ни у одной пары процессоров нет общей памяти. Кроме того, не поддерживается вызов страниц по требованию, поскольку для размещения страниц нет локальных дисков. Вместо этого в системе имеются 11 524 узла ввода-вывода, которые соединяются с дисками и другими периферийными устройствами.

Несмотря на исключительные размеры системы, она довольно проста и в ней не используются какие-либо особенные технологии, за исключением, разве что, чрезвычайно плотного размещения узлов. Это не случайно, так как основными целями были надежность и доступность. Соответственно, весьма тщательно были спроектированы системы питания, охлаждения, кабельные системы и т. д., все это позволило поднять среднее время наработки на отказ до 10 дней.

Для подключения всех микросхем требуется масштабируемая и высокопроизводительная схема соединений. В качестве топологии был выбран трехмерный тор размером $72 \times 32 \times 32$. Таким образом, каждой микросхеме требуются 6 линий связи: 2 для соседей, логически расположенных сверху и снизу, 2 для соседей с севера и юга, 2 для соседей с запада и востока (см. соответствующие

обозначения на рис. 8.32). Конструктивно, каждая стойка на 1024 узла образует тор размером 8 × 8 × 16. Пара соседних стоек соединяется в тор размером 8 × 8 × 32. Четыре пары стоек из одного ряда образуют тор размером 8 × 32 × 32, и наконец, все 8 рядов дают тор размером 72 × 32 × 32.

Таким образом, все соединения являются двухточечными и работают на скорости 3,4 Гбит/с. Так как от каждого из 73 728 узлов к «следующим по порядку» узлам идут три линии связи, по одной на каждое измерение, общая пропускная способность системы составляет 752 Тбит/с. Информационная емкость книги, которую вы держите в руках, включая рисунки в формате Encapsulated PostScript, составляет около 300 млн бит, таким образом, BlueGene/P может за секунду передать внутри себя до 2,5 млн ее копий. Куда девать эти копии, и кто мог бы их прочесть — эти вопросы мы оставляем читателю.

Взаимодействие в трехмерном торе поддерживается в форме **виртуальной сквозной маршрутизации** (virtual cut through routing). Этот подход в чем-то напоминает коммутацию с сохранением и продвижением пакетов (см. раздел «Сопроцессоры»), за исключением того, что перед дальнейшим продвижением по линии связи пакеты целиком не сохраняются. Как только очередной байт пакета прибывает на транзитный узел, он передается вдоль маршрута далее, не дожидаясь получения всего пакета. Допускается как динамическая (адаптивная), так и статическая (фиксированная) маршрутизация. Для реализации виртуальной сквозной маршрутизации на микросхеме имеются несколько специализированных устройств.

В дополнение к основному трехмерному тору, обеспечивающему обмен данными, есть и другие коммуникационные сети. Вторая (коллективная) сеть имеет древовидную структуру. В системах с высокой степенью параллелизма, таких как BlueGene/P, для выполнения многих операций требуется участие всех узлов. В качестве примера рассмотрим задачу поиска наименьшего из 65 536 значений, каждое из которых хранится в отдельном узле. Если все узлы связаны в древовидную структуру, каждые два узла могут отправить вышестоящему узлу свои значения, тот может выбрать из них меньшее и передать его выше. При таком подходе в корневой узел попадает лишь необходимый минимум информации (представьте, если бы каждый из 65 535 узлов непосредственно передал сообщение корневому узлу).

Третья (барьерная) сеть используется для глобальных приостановок и прерываний. Некоторые алгоритмы требуют поэтапного выполнения, когда каждый узел, закончив свой этап, не переходит к следующему, а ожидает, пока тот же этап закончат все остальные. Особая барьерная сеть позволяет программно задавать эти этапы и приостанавливать вычисления на всех процессорах, завершивших свой этап раньше остальных. Когда все процессоры завершают свой этап, вычисления продолжаются. Та же барьерная сеть используется для прерываний.

Четвертая и пятая сети используют технологию Ethernet 10 Гбит. Одна из них соединяет узлы ввода-вывода с файловыми серверами, не входящими в систему BlueGene/P, а также с Интернетом; другая используется для отладки системы.

На каждом вычислительном и коммуникационном узле работает специализированная малая операционная система, поддерживающая одного пользователя и один процесс. Процесс может иметь до четырех программных потоков, по одному на каждый процессор в узле. Эта простая структура была выбрана за ее высокую производительность и надежность.

Для повышения надежности прикладная программа может создать точку сохранения, вызвав библиотечную процедуру. После того как в сети закончится передача всех еще не переданных сообщений, можно создать глобальную точку сохранения, чтобы при сбое системы задание можно было запустить с этой точки, а не с самого начала. Узлы ввода-вывода работают под управлением традиционной ОС Linux и поддерживают многозадачность.

Сейчас ведется работа над следующим поколением системы BlueGene, названному BlueGene/Q. Предполагается, что эта система будет запущена в эксплуатацию в 2012 году, будет содержать 18 процессоров на микросхему, а также будет поддерживать параллельную многопоточность. Эти две особенности значительно увеличат производительность системы в количестве команд на цикл. Предполагается, что система достигнет скорости 20 петафлоп/с. Дополнительную информацию о BlueGene/L можно найти в [Adigaetal, 2002; Alam et al., 2008; Almasi et al., 2003a и 2003b; Blumrich et al., 2005; IBM, 2008].

Red Storm

В качестве второго примера систем MPP рассмотрим разработанную в национальной лаборатории Sandia машину Red Storm (также называемую Thor's Hammer). Лаборатория Sandia работает под управлением Lockheed Martin и выполняет секретные и несекретные задания департамента энергетики США. Среди секретных работ можно назвать моделирование ядерных взрывов, требующее очень интенсивных вычислений.

Sandia давно в этом бизнесе и многие годы обладает самыми мощными суперкомпьютерами. В течение десятилетий здесь отдавалось предпочтение векторным суперкомпьютером, но в определенный момент, благодаря развитию технологии и изменениям в экономике, на смену им стали приходить MPP-машины. Начиная с 2002 года использовавшаяся тогда MPP-машина под названием ASCI Red стала все сильнее «пробуксовывать». Хотя в ней было 9460 узлов, вместе они предлагали лишь 1,2 Тбайт ОЗУ и 13,5 Тбайт дискового пространства, к тому же система в целом с трудом поддерживала производительность 3 терафлоп/с. Поэтому в 2002 году в Sandia решили заменить ASCI Red, выбрав в качестве долгосрочного поставщика суперкомпьютеров компанию Cray Research.

Новая система была поставлена в августе 2004 года, что очень быстро для разработки и реализации столь большой машины. Причина такой оперативности состоит в том, что мультикомпьютер Red Storm построен почти исключительно из обычных имеющихся в продаже компонентов. Исключение составляет только специализированная микросхема, используемая для маршрутизации. В 2006 году система была переоснащена новыми процессорами; эта версия описывается ниже.

Для Red Storm был выбран процессор двухъядерный Opteron 2,4 ГГц производства компании AMD. Этот выбор обусловили несколько его ключевых характеристик. Первая — поддержка трех режимов работы. В унаследованном режиме на этом процессоре без всякой модификации можно выполнять обычные программы, рассчитанные на Pentium. В режиме совместимости операционная система работает как 64-разрядная и может адресовать до 264 байт памяти, в то время как прикладные программы являются 32-разрядными. Наконец, в 64-разрядном режиме машина целиком становится 64-разрядной и может адресовать все 64-разрядное адресное пространство. Причем в 64-разрядном режиме одно-

временно могут работать и 32-разрядные, и 64-разрядные программы, что упрощает обновление системы.

Еще одной ключевой характеристикой Opteron является тщательная проработка вопросов пропускной способности памяти. В последние годы процессоры становились все быстрее и быстрее, заметно опережая в этой гонке память. В результате, в случае кэш-промаха в кэше второго уровня время обращения к памяти значительно возрастает. Инженеры AMD установили в процессор Opteron контроллер памяти, работающий на частоте процессора, а не на частоте шины памяти, что повышает производительность памяти. Контроллер может работать с восемью модулями DIMM по 4 Гбайт каждый, что дает максимальный объем памяти в 32 Гбайт. В системе Red Storm для каждого процессора Opteron устанавливается 2–4 Гбайт, но нет сомнений, что со временем, по мере удешевления памяти, это значение будет увеличено. Другая возможность повышения производительности системы — замена процессоров Opteron двухъядерными моделями, что теоретически должно удвоить вычислительную мощность.

Каждому процессору Opteron выделяется собственный специализированный сетевой процессор под названием **Seastar** производства IBM. Это критически важный элемент системы, так как практически весь обмен информацией между процессорами происходит через сеть Seastar. Без высокоскоростной коммуникационной сети, функционирование которой поддерживают эти микросхемы, система быстро бы «утонула» в данных.

Хотя процессоры Opteron — это обычные процессоры, имеющиеся в продаже, в Red Storm они устанавливаются в собственные специализированные платы (рис. 8.36). На каждой такой плате располагаются 4 процессора Opteron,

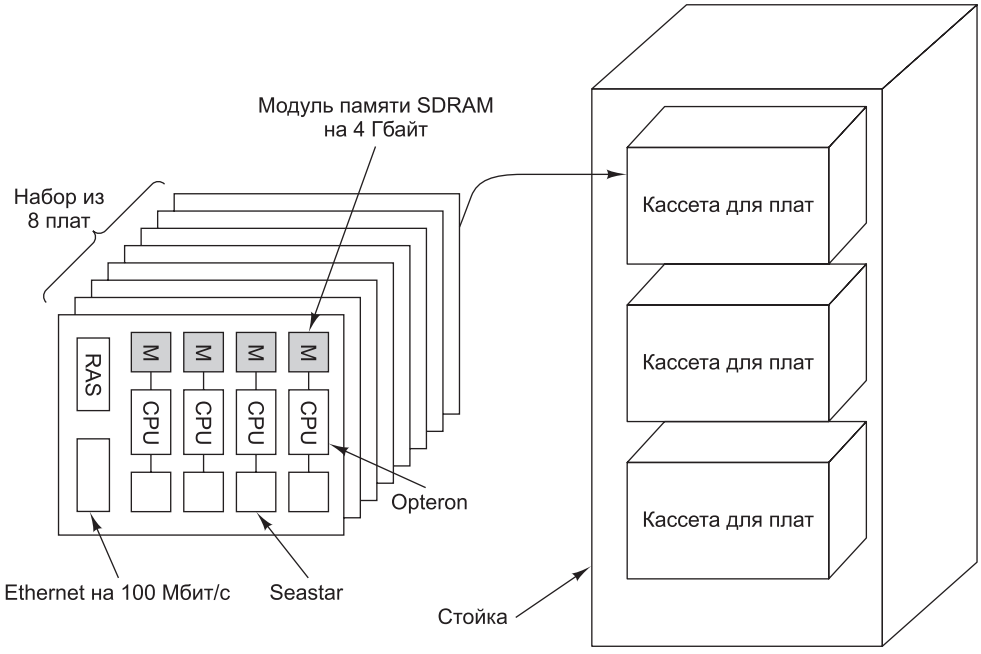


Рис. 8.36. Размещение компонентов Red Storm

оперативная память на 4 Гбайт, 4 процессора Seastar, процессор RAS (Reliability, Availability and Service — надежность, работоспособность и удобство эксплуатации), микросхема Ethernet на 100 Мбит/с.

Набор из восьми плат устанавливается в панель и вставляется в кассету. В каждой стойке есть три таких кассеты, что позволяет установить 96 процессоров Opteron, а также необходимые источники питания и систему охлаждения. Вся система состоит из 108 стоек, что дает 10 368 процессоров с модулями памяти SDRAM объемом 10 Тбайт. Каждый процессор имеет доступ только к своему модулю SDRAM, общей памяти нет. Теоретическая вычислительная мощность системы равна 41 терафлоп/с.

Для взаимосвязи отдельных центральных процессоров Opteron служат специализированные маршрутизаторы Seastar, по одному маршрутизатору на процессор. Они соединены друг с другом в трехмерный тор размером $27 \times 16 \times 24$, в каждом узле которого находится один маршрутизатор. У каждого маршрутизатора 7 двусторонних высокоскоростных (24 Гбит/с) линий связи. Шесть из них ведут к соседям: на север, восток, юг, запад, вверх и вниз, а еще одна соединяет маршрутизатор с процессором Opteron. Время передачи между соседними узлами решетки составляет 2 мкс. Для прохождения всего набора вычислительных узлов требуется 5 мкс. Еще одна сеть построена на основе технологии Ethernet со скоростью 100 Мбит/с и служит для обслуживания и поддержки системы.

В дополнение к 108 вычислительным стойкам в системе имеются 16 стоек для процессоров ввода-вывода и обслуживания. В каждом из них установлено 32 процессора Opteron. Из этих 512 процессоров 256 отвечают за ввод-вывод и 256 — за обслуживание. Остальное место занимают диски, организованные в RAID-массивы уровня 3 и 5, каждый с диском четности и диском «горячей» замены. Общий объем дискового пространства составляет 240 Тбайт. Объединенная производительность дисковой системы равна 50 Гбайт/с.

Система при помощи механических переключателей разбивается на две части, секретную и несекретную, которые при необходимости можно разъединять. Из общего числа процессоров 2688 всегда находятся в секретной секции, еще 2688 — всегда в несекретной. Остальные 4992 вычислительных процессора можно переключать между секциями, как показано на рис. 8.37. Все процессоры Opteron из секретной секции имеют по 4 Гбайт памяти, все остальные — по 2 Гбайт. Процессоры ввода-вывода и обслуживания поделены между секциями.

Вся система располагается в отдельном здании площадью 2000 м^2, спроектированное и построенное так, чтобы в будущем при необходимости можно было бы расширить систему до 30 000 процессоров. Энергопотребление вычислительных узлов составляет 1,6 МВт, еще 1 МВт потребляют диски. Вместе с системой вентиляции и кондиционирования воздуха общее энергопотребление равно 3,5 МВт.

Стоимость аппаратного и программного обеспечения компьютера равна 90 млн долларов, здание и вентиляция стоят еще 9 млн, что в общем дает немногим меньше 100 млн долларов, хотя часть этих денег составляют единовременные расходы на саму разработку. То есть если вы хотите получить точную копию, приготовьте порядка 60 млн долларов. К тому же в Cray рассчитывают продавать уменьшенные версии этой системы для правительственных и частных организаций под названием X3T.

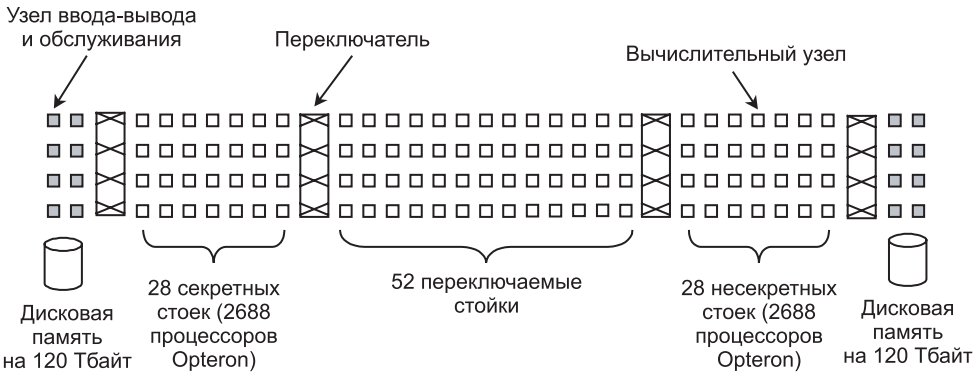


Рис. 8.37. Система Red Storm, вид сверху

Вычислительные узлы работают под управлением облегченного ядра, названного **Catamount** («дикая кошка»). Узлы ввода-вывода управляются обычной ОС Linux с небольшими дополнениями, связанными с поддержкой интерфейса MPI (см. далее в этой главе). В RAS-узлах работает урезанная версия Linux. На Red Storm можно запускать имеющиеся в изобилии программы для ASCI Red, среди которых есть программы выделения процессоров, планировщики, MPI-библиотеки, математические библиотеки, прикладные программы.

Для такой большой системы вопросы надежности выходят на первый план. На каждой плате имеется RAS-процессор, предназначенный для обслуживания системы, а также специализированные аппаратные средства. Целью разработчиков было обеспечить показатель MTBF (Mean Time Between Failures — среднее время наработки на отказ), равный 50 часам. У аппаратного обеспечения ASCI Red этот показатель был равен 900 часам, но все портила операционная система — полный отказ системы случался каждые 40 часов. И хотя новая аппаратура намного надежнее старой, слабым местом по-прежнему остаются программы.

Более подробные сведения о Red Storm можно найти в [Brightwell et al., 2005, 2010].

Сравнение систем BlueGene/P и Red Storm

Системы BlueGene/L и Red Storm, будучи схожими во многих отношениях, имеют немало отличий, поэтому сравнить их друг с другом достаточно интересно. В табл. 8.6 приведены значения некоторых их ключевых параметров.

Таблица 8.6. Сравнение систем BlueGene/7 и Red Storm

Параметр	BlueGene/L	Red Storm
Центральный процессор	32-разрядный PowerPC	64-разрядный Opteron
Частота	850 МГц	2,4 ГГц
Количество вычислительных процессоров	294 912	20 736
Количество процессоров на плате	128	8

Параметр	BlueGene/L	Red Storm
Количество процессоров в стойке	4096	192
Количество вычислительных стоек	72	108
Производительность (терафлопов в секунду)	1000	124
Объем памяти на одном процессоре	512 Мбайт	2–4 Гбайт
Общий объем памяти	144 Тбайт	10 Тбайт
Маршрутизатор	PowerPC	Seastar
Количество маршрутизаторов	73 728	10 368
Топология	Тор размером $72 \times 32 \times 32$	Тор размером $27 \times 16 \times 24$
Дополнительные сети	Gigabit Ethernet	Fast Ethernet
Возможность деления на секции	Нет	Есть
ОС для вычислительных узлов	Специализированная	Специализированная
ОС ввода-вывода	Linux	Linux
Производитель	IBM	Cray Research
Стоимость	Высокая	Высокая

Эти две машины построены примерно в одно и то же время, поэтому различия между ними определяются не технологией, а склонностями разработчиков, а также, в некоторой степени, различиями между компаниями IBM и Cray. Система BlueGene/P с самого начала была спроектирована как коммерческая машина, ориентированная на продажу биотехнологическим, фармацевтическим и другим компаниям. В противоположность этому, система Red Storm была построена по индивидуальному заказу лаборатории Sandia, к тому же компания Cray планирует выставлять на продажу уменьшенные версии системы.

Подход IBM очень прост: из существующих ядер построить, хотя и специализированную, но низкоскоростную и дешевую в массовом производстве микросхему, а затем очень большое количество этих микросхем объединить не слишком скоростной сетью. Другой, но столь же понятный подход выбрали в Sandia: взять наиболее мощный из имеющихся в продаже 64-разрядный процессор, снабдить его высокоскоростным специализированным маршрутизатором и большим объемом памяти, получив гораздо более мощные вычислительные узлы, чем узлы BlueGene/P. Таких узлов потребуется гораздо меньше, поэтому и обмен информацией между ними будет происходить, соответственно, быстрее.

Каждое решение повлияло на размещение элементов по-своему. Благодаря тому, что в IBM разработали специализированную микросхему, объединившую процессор и маршрутизатор, удалось добиться более высокой плотности упаковки — 1024 процессора в стойке. В Sandia на каждый узел установили обычный

массовый процессор и память объемом 2–4 Гбайт, поэтому в стойке удалось разместить только 96 вычислительных процессоров. Как следствие, Red Storm занимает больше места и потребляет больше энергии, чем BlueGene/P.

В экзотическом мире национальных исследовательских лабораторий главным критерием является производительность. По этому параметру BlueGene/P выигрывает, так как производительность этой системы составляет 1000 терафлоп/с против 124 терафлоп/с у Red Storm. Но нельзя забывать, что конструкция Red Storm расширяема, поэтому Sandia, вероятно, сможет поднять производительность за счет ввода дополнительных процессоров Opteron. Однако и IBM в состоянии ответить на это некоторым увеличением тактовой частоты (действительно, частота 850 МГц не слишком впечатляет). Другими словами, MPP-суперкомпьютеры еще не подошли к физическому пределу своей мощности и будут развиваться еще долгие годы.

Кластерные вычисления

Другой вариант мультикомпьютера — **кластерный компьютер** [Anderson et al., 1995; Martin et al., 1997]. Как правило, кластер состоит из нескольких сотен или тысяч связанных сетью персональных компьютеров или рабочих станций, причем к сети они подключаются через обычную сетевую плату. Различие между MPP и кластером такое же, как между мэйнфреймом и персональным компьютером. У обоих есть процессор, ОЗУ, диски, операционная система и т. д. Но в мэйнфрейме все это (за исключением, может быть, операционной системы) работает гораздо быстрее, и из-за этого применяются и управляются они совершенно поразному. То же самое можно сказать о MPP и кластерах.

Еще несколько лет назад взаимодействие между элементами, образующими MPP, происходило гораздо быстрее, чем между машинами, составляющими кластер. Однако с появлением на рынке высокоскоростных сетей этот разрыв стал сходить «на нет». Вероятно, кластеры постепенно вытеснят MMP-машины, подобно тому как персональные компьютеры вытеснили мэйнфреймы, которые применяются теперь только в узкоспециализированных областях. Основной нишей для систем MMP останутся дорогостоящие суперкомпьютеры, в которых главное — производительность, а стоимость уходит на второй план.

Существуют множество видов кластеров, два из которых доминируют: централизованные и децентрализованные. Централизованным называют кластер рабочих станций или персональных компьютеров, смонтированных в большую конструкцию в пределах одной комнаты. Иногда они располагаются более компактно, чем обычно, чтобы сократить физические размеры и длину кабеля. Как правило, все входящие в кластер машины гомогенны и не имеют никаких периферийных устройств, кроме сетевых плат и, возможно, дисководов. Гордон Белл (Gordon Bell), разработчик PDP-11 и VAX, назвал их **безголовыми рабочими станциями**, намекая на то, что у этих машин нет владельцев.

Децентрализованные кластеры состоят из рабочих станций или персональных компьютеров, разбросанных в пределах здания или кампуса. Большинство из них простаивают много часов в день, особенно ночью. Обычно они связаны локальной сетью. Они гетерогенны и имеют полный набор периферийных устройств, хотя кластер с тысячью мышей ничем не лучше кластера вообще без мышей.

Самое важное то, что многие входящие в кластер машины имеют владельцев, каждый из которых души не чает в своей машине и не слишком лояльно относится к тому, что какой-то астроном пытается вовлечь ее в вычисления, связанные с теорией большого взрыва. Если же использовать для организации кластера только бездействующие на данный момент машины, обязательно нужен какой-то механизм миграции заданий, чтобы освободить машину, когда она понадобится своему владельцу. Хотя проблема миграции заданий вполне решаема, решение требует дополнительного усложнения программного обеспечения.

Кластеры зачастую невелики — в пределах от дюжины до, возможно, 500 компьютеров. Тем не менее можно построить очень большой кластер из обычных ПК. Компания Google предложила для этого интересный способ, который мы здесь рассмотрим.

Google

Google — популярная система поиска информации в Интернете. Своей популярностью она обязана простоте интерфейса и малому времени отклика, хотя внутреннее устройство Google можно назвать каким угодно, но только не простым. С точки зрения поисковой системы задача состоит в том, чтобы проиндексировать и сохранить всю Всемирную паутину (а это более 40 млрд страниц), а затем находить среди сохраненной информации нужную страницу за 0,5 секунды, обслуживая десятки тысяч запросов в секунду, круглосуточно приходящих со всех концов света. В дополнение к этому, система никогда не должна отключаться, даже в случае природных катаклизмов, перебоев в электропитании и работе сети, аппаратных и программных сбоев. Разработка клона Google — определенно не упражнение для читателя.

Как же Google все это делает? Функционирование Google обеспечивают множество информационных центров по всему миру. Это не только делает возможной подмену на случай внезапного наводнения или землетрясения. При обращении к адресу www.google.com анализируется IP-адрес отправителя запроса, и далее браузер общается уже только с ближайшим к нему информационным центром.

Каждый информационный центр подключается к Интернету как минимум одной оптоволоконной линией OC-48 (2,488 Гбит/с), по которой поступают запросы и отправляются обратно ответы. Кроме того, имеется дополнительная линия OC-12 (622 Мбит/с) к резервному поставщику услуг на случай перерыва в работе основного. Чтобы шоу могло продолжаться и при перебоях в электроснабжении, во всех центрах имеются источники бесперебойного питания и аварийные дизельные генераторы. Таким образом, во время природных катаклизмов работа Google не нарушится, хотя производительность и снизится.

Чтобы лучше понять, почему в Google была выбрана именно такая архитектура, полезно познакомиться с механизмом обработки запроса, пришедшего в информационный центр. Поступающий запрос (шаг *1* на рис. 8.38) переправляется распределителем нагрузки к одному из многочисленных обработчиков запросов (*2*), а также, параллельно, в систему проверки правописания (*3*) и сервер контекстной рекламы (*4*). Параллельно выполняется поиск запрошенного слова на индексных серверах (*5*), на которых хранятся записи о каждом слове в Сети. В каждой такой записи перечислены все содержащие это слово документы (это

могут быть веб-страницы, PDF-файлы, презентации PowerPoint и т. д.). Ссылки в этих списках расположены в соответствии с рейтингом страницы — параметром, который вычисляется по сложной формуле. Принцип вычисления рейтинга держится в тайне, но известно, что большое значение имеет количество ссылок на страницу и рейтинги ссылающихся на нее страниц.

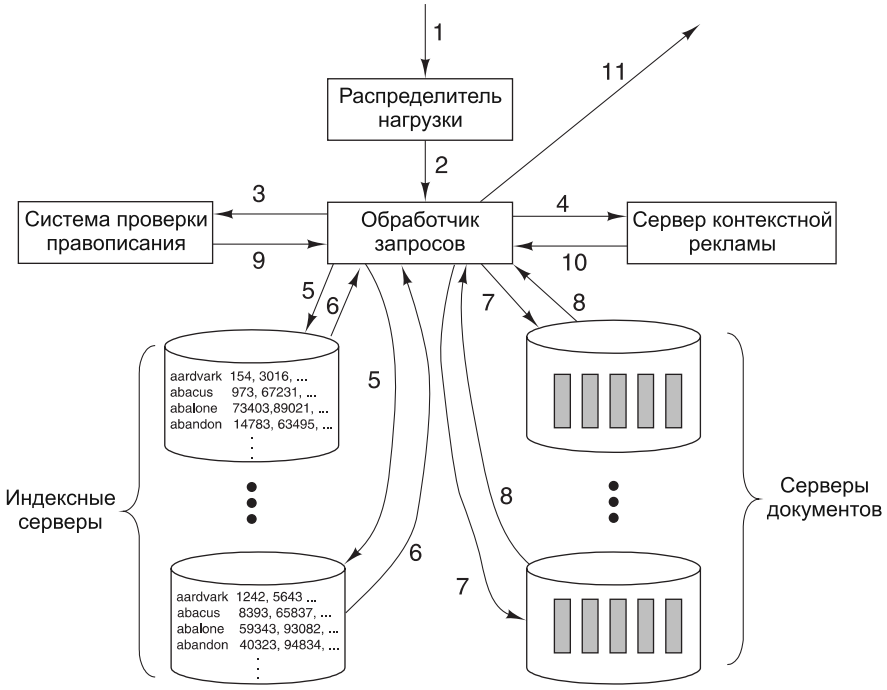


Рис. 8.38. Обработка запроса в Google

Для повышения производительности индекс разбит на **сегменты**, поиск в которых ведется параллельно. Согласно этой идее, сегмент *1* содержит все слова из индекса, и каждому слову сопоставлены идентификаторы *n* первых по рейтингу страниц. Сегмент *2* содержит все слова и идентификаторы *n* следующих по рейтингу страниц и т. д. По мере роста Сети, каждый из этих сегментов можно дополнительно разделить на несколько частей так, что в первой части будут первые *k* слов, во второй — следующие *k* и т. д. Это позволяет достигать еще большего параллелизма при поиске.

Индексные серверы возвращают наборы идентификаторов документов (*6*), которые затем комбинируются в соответствии с логикой запроса. Например:

```
+digita +kapibara +dance
```

При таком запросе на следующий шаг попадут идентификаторы только тех документов, которые имеются во всех трех наборах. На этом шаге Google обращается к самим документам (*7*), извлекая из них названия, ссылки, а также фрагменты текста, окружающие запрошенные слова. Копии многих документов Сети хранятся на серверах документов всех информационных центров, на настоящее время их объем достигает сотен терабайтов. Для ускорения параллельного

поиска документы также поделены на сегменты. В итоге, хотя для обработки запроса не требуется считывать все содержимое Сети (и обрабатывать десятки терабайтов индексов), при обслуживании рядового запроса все же приходится «переворошить» не менее 100 Мбайт данных.

После того как результаты возвращаются обработчику запроса (*8*), они объединяются в соответствии с рейтингом страниц. Добавляется информация о возможных ошибках правописания, если они обнаружены (*9*), и контекстная реклама (*10*). Включение в результаты запроса тех или иных ключевых слов, купленных рекламодателями (например, «гостиница», или «camcoder»), — это то, за счет чего Google зарабатывает деньги. Наконец, результаты оформляются в формате HTML (HyperText Markup Language — язык разметки гипертекста) и передаются пользователю в виде обычной веб-страницы.

Теперь, опираясь на полученные знания, мы можем изучить архитектуру Google. Большинство компаний, сталкиваясь с необходимостью поддерживать громадную и высоконадежную базу данных с колоссальным количеством транзакций, приобретают самое быстрое и надежное оборудование, имеющееся на рынке. В Google поступили прямо наоборот. Они купили дешевые персональные компьютеры со средней производительностью. Много компьютеров. Объединив эти машины, они построили самый большой в мире кластер из обычных компонентов. Главный принцип, лежащий в основе этого решения, прост — оптимизация отношения цена/производительность.

Корни принятого решения лежат в экономике: обычные персональные компьютеры достаточно дешевы. Для высококлассных серверов это не так, а для больших мультипроцессоров — вдвойне не так. К примеру, производительность мощного сервера может в 2–3 раза превышать производительность среднего ПК, а вот стоит он дороже обычно не в 2–3, а в 5–10 раз.

Конечно же, дешевый персональный компьютер гораздо менее надежен, чем лучшие модели серверов, но ведь и серверы иногда «падают». Поэтому программное обеспечение Google написано так, чтобы надежно работать на ненадежном аппаратном обеспечении. Имея в своем распоряжении отказоустойчивое программное обеспечение, уже не имеет большого значения, какова интенсивность отказов, 0,5 или 2 % в год. Опыт Google показывает, что за год ломаются 2 % всех компьютеров. Более половины отказов вызваны жесткими дисками, следующей причиной являются блоки питания, а за ними следуют микросхемы памяти. Процессоры, после обкатки, вообще не ломаются. В действительности, основной причиной сбоев является не аппаратное, а программное обеспечение. Поэтому первой реакцией на ошибку является перезагрузка, которая в большинстве случаев решает проблему (это можно назвать электронным аналогом рецепта «две таблетки аспирина на ночь»).

В типичном современном ПК, используемом в Google, установлен процессор Pentium с тактовой частотой 2 ГГц, 4 Гбайт оперативной и 2 Тбайт дисковой памяти. Возможно, примерно на такой же машине ваша бабушка изредка просматривает электронную почту. Особого внимания заслуживает только микросхема Ethernet. Ее нельзя назвать произведением искусства, зато она очень дешево стоит. Компьютеры размещаются в корпусах высотой 1u (это около 5 см) и устанавливаются в вертикальные стойки, по 40 штук с передней и задней сторон. В одной стойке, таким образом, устанавливаются 80 машин, которые подключа-

ются к Ethernet при помощи коммутатора внутри стойки. Все стойки в одном информационном центре также подключены к Ethernet через коммутатор, а для живучести при сбоях имеются два избыточных коммутатора.

Структура типичного информационного центра Google показана на рис. 8.39. Данные с высокоскоростной оптоволоконной линии OC-48 поступают на два 128-портовых Ethernet-коммутатора. Аналогично к ним подключена и резервная линия OC-12. Для подключения входных каналов применяется специальная плата, поэтому они не занимают порты Ethernet-коммутаторов. Из каждой стойки выходит четыре Ethernet-линии, две к коммутатору, показанному на рисунке слева, и две к правому. Благодаря этому система может пережить отказ любого из двух коммутаторов. Благодаря наличию четырех линий для потери связи со стойкой необходимо, чтобы либо вышли из строя все четыре линии, либо чтобы вышли из строя две линии и коммутатор. Имея пару коммутаторов на 128 портов и стойки с четырьмя линиями, можно соединить в сеть 64 стойки. Если считать, что в стойке 80 компьютеров, это дает суммарно 5120 машин, хотя, конечно же, никто не требует, чтобы в стойке было именно 80 машин, да и у коммутаторов может быть больше 128 портов. Просто это — весьма характерные для кластера Google значения.

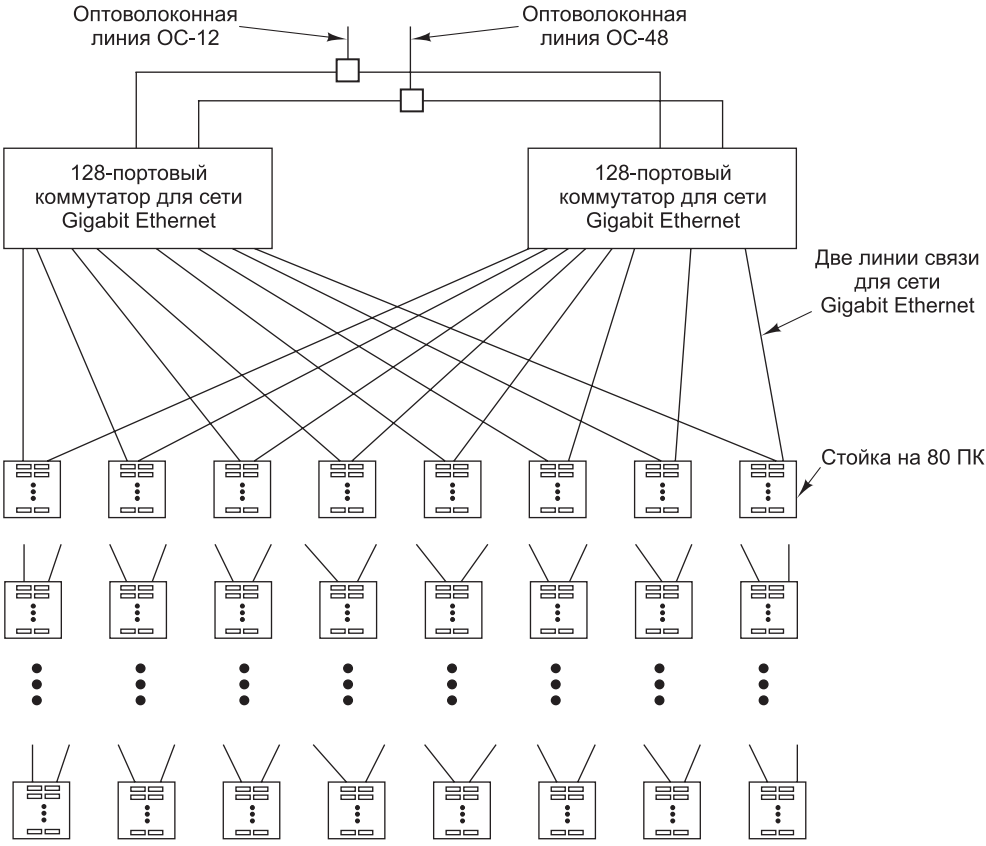


Рис. 8.39. Типичный кластер Google

Большое значение имеет также удельное энергопотребление на единицу площади. Типичный персональный компьютер потребляет около 120 Вт, что дает 10 кВт на стойку. Чтобы обслуживающий персонал мог устанавливать в стойку и извлекать из стойки компьютеры, для нее требуется не менее 3 $м^2$ пространства. Таким образом, удельное энергопотребление составляет 3000 $Вт/м^2$. Обычно информационные центры проектируются в расчете на энергопотребление от 600 до 1200 $Вт/м^2$, поэтому требуется предпринимать специальные меры для охлаждения.

В Google хорошо усвоили три правила создания и использования крупных веб-серверов, которые неплохо здесь воспроизвести.

1. Любые компоненты ломаются, и это надо учитывать.
2. Для повышения пропускной способности и доступности все должно дублироваться.
3. Необходимо оптимизировать соотношение цена/производительность.

Первый пункт фактически означает, что программное обеспечение должно быть отказоустойчивым. Даже лучшее оборудование рано или поздно сломается, если его достаточно много, и надо программно учитывать эту возможность. Система такого размера должна переживать отказы, даже если происходят они несколько раз в неделю.

Второй пункт указывает на то, что и программное, и аппаратное обеспечение должны обладать избыточностью. Это не только делает систему отказоустойчивой, но и повышает пропускную способность. В случае Google сами компьютеры, их диски, кабели, блоки питания и коммутаторы многократно дублируются. Более того, дублируются в пределах одного центра сегменты индексов и документов. Дублируются и сами информационные центры.

Третий пункт является следствием первых двух. Если система должным образом реагирует на сбои, глупо покупать дорогие компоненты, такие как RAID-массивы или SCSI-диски. Даже они ломаются, а тратить в десть раз больше, чтобы вдвое снизить интенсивность отказов — плохая идея. Лучше купить в десять раз больше оборудования и предусмотреть возможность отказов. В конце концов, чем больше оборудования, тем выше производительность (когда оборудование работает).

Более подробные сведения о Google ищите в [Barroso et al., 2003; Ghemawa et al., 2003].

Коммуникационное программное обеспечение для мультикомпьютеров

Для программирования мультикомпьютера требуется специальное программное обеспечение (обычно это библиотеки), позволяющее обеспечить взаимодействие между процессами и синхронизацию. Отметим, что в большинстве случаев программные пакеты предназначаются и для MPP-машин, и для кластеров, поэтому приложения являются переносимыми между платформами.

В системах передачи сообщений два и более процессов работают независимо друг от друга. Например, один из процессов может генерировать данные, а другой (или другие) — их потреблять. Если у отправителя есть еще данные, нет никакой

гарантии, что получатель (получатели) готов принять эти данные, поскольку каждый процесс работает по собственной программе.

В большинстве систем передачи сообщений используются два примитива `send` и `receive`, но возможны и другие варианты семантики. Тремя основными вариантами являются:

- синхронная передача сообщений;
- буферизуемая передача сообщений;
- неблокирующая передача сообщений.

Если при **синхронной передаче сообщений** отправитель выполняет операцию `send`, а получатель еще не выполнил операцию `receive`, отправитель блокируется (приостанавливается) до тех пор, пока получатель не выполнит операцию `receive`, а в это время сообщение копируется. Когда управление возвращается отправителю, он уже знает, что отправленное сообщение получено. Этот метод имеет простую семантику и не требует буферизации. Но у него есть серьезный недостаток: отправитель блокируется до тех пор, пока получатель не примет сообщение и подтвердит его прием.

В случае **буферизуемой передачи сообщений** отправленное сообщение временно где-либо сохраняется (например в почтовом ящике), пока получатель не будет готов его оттуда забрать. При таком подходе отправитель может продолжать работу и после выполнения операции `send`, даже если получатель в этот момент занят. Поскольку сообщение уже отправлено, отправитель может сразу же снова использовать буфер сообщений. Такая схема сокращает время запаздывания. Однако никаких гарантий получения сообщения нет. Даже при надежной коммуникационной системе всегда есть вероятность, что из-за сбоя получатель не смог получить сообщение.

При **неблокирующей передаче сообщений** отправитель, сделав вызов, может сразу продолжать работу. Единственная задача библиотеки — сообщить операционной системе, чтобы та обработала вызов, когда у нее появится время. В результате отправитель вообще не блокируется. Недостаток этого метода состоит в том, что когда отправитель продолжает работу после выполнения операции `send`, он не может снова использовать буфер сообщений, так как есть вероятность, что сообщение еще не отправлено. Отправитель каким-то образом должен определять, что он может снова использовать буфер. Например, можно опрашивать систему. Другой вариант — при освобождении буфера выполнять системное прерывание. Однако оба варианта требуют сложного программного обеспечения.

Далее мы рассмотрим популярную систему передачи сообщений, которая применяется во многих мультикомпьютерах.

MPI

Еще несколько лет назад пакет **PVM** (Parallel Virtual Machine — **параллельная виртуальная машина**) считался самым популярным пакетом для обмена информацией между мультикомпьютерами [Geist et al., 1994; Sunderram,. 1990]. Однако в настоящее время он почти повсеместно вытеснен пакетом **MPI** (Message-Passing Interface — **интерфейс передачи сообщений**). Пакет MPI гораздо сложнее, чем PVM; он поддерживает намного больше библиотечных вызовов и намного больше параметров для каждого вызова. Первая версия MPI, которая сейчас на-

зывается MPI-1, в 1997 году была дополнена второй версией, MPI-2. Далее мы кратко расскажем о MPI-1, а затем посмотрим, что нового появилось в MPI-2. Более подробную информацию об MPI см. в [Gropp et al., 1994; Snir et al., 1996].

Пакет MPI-1 в отличие от PVM никак не связан с созданием процессов и управлением процессами. Создавать процессы должен сам пользователь путем локальных системных вызовов. После создания процессы организуются в группы, которые уже не изменяются. Именно с этими группами работает MPI.

В основе MPI лежат четыре концепции: коммуникаторы, типы передаваемых данных, коммуникационные операции и виртуальные топологии. **Коммуникатор** — это группа процессов плюс контекст. Контекстом называют метку, которая идентифицирует что-либо (например, фазу выполнения). В процессе отправки и получения сообщений контекст может использоваться для того, чтобы не перепутать сообщения, не имеющие отношения друг к другу.

Типы передаваемых в сообщениях данных могут быть разными, в том числе символами, короткими, обычными и длинными целыми, числами с плавающей точкой обычной и двойной точности и т. д. Кроме того, из существующих типов данных можно строить новые.

MPI поддерживает множество коммуникационных операций. Вот как выглядит операция для отправки сообщений:

```
MPI_Send(буфер, число_элементов, тип_данных, получатель, тег, коммуникатор)
```

В этом вызове *получателю* передается содержимое *буфера*, в котором находятся элементы типа *тип_данных* в количестве *число_элементов*. Поле *тега* — это метка сообщения; получатель имеет возможность получать только те сообщения, в которых имеется данный тег. Последнее поле (*коммуникатор*) показывает, к какой группе процессов относится получатель (*получатель* — это просто индекс в списке процессов из определенной группы). Аналогичный вызов для получения сообщения выглядит так:

```
MPI_Recv(&буфер, число_элементов, тип_данных, отправитель,
  тег, коммуникатор, &статус)
```

Этот вызов говорит о том, что получатель ждет от *отправителя* сообщение типа *тип_данных* с заданным *тегом*.

MPI поддерживает четыре базовых коммуникационных режима. Первый режим — синхронный. В нем отправитель не может начать передачу данных, пока получатель не вызовет `MPI_Recv`. Второй режим — буферизация. В этом режиме ограничение первого режима не действует. Третий режим — стандартный. Он зависит от конкретной реализации, то есть может быть реализован либо синхронный режим, либо режим буферизации. Четвертый режим — готовность. Здесь, как и в синхронном режиме, отправитель требует, чтобы получатель был доступен, но без проверки. Каждый из этих примитивов имеет блокирующую и неблокирующую версии, что в сумме дает 8 примитивов. Получение может быть только в двух вариантах: блокирующим и неблокирующим.

MPI поддерживает разнообразные формы коллективного взаимодействия. В любой форме такого взаимодействия все процессы в группе должны делать вызов с совместимыми параметрами. В противном случае возникает ошибка. Например, в типичной форме коллективного взаимодействия процессы могут быть организованы в виде древовидной структуры, в которой данные передаются

от листьев к корню, подвергаясь на каждом шаге определенной обработке (например, суммирование или поиск максимума).

Четвертая базовая концепция MPI — **виртуальная топология** процессов (дерево, кольцо, решетка, тор и т. д.). Такая организация процессов обеспечивает возможность именования коммуникационных маршрутов и упрощает взаимодействие.

В MPI-2 была добавлена поддержка динамических процессов, удаленного доступа к памяти, неблокирующего коллективного взаимодействия, масштабируемого ввода-вывода, обработки в режиме реального времени и многого другого. В научном сообществе годами шла «война» между сторонниками MPI и PVM. Сторонники PVM утверждали, что эту систему проще изучать и легче использовать. Почитатели MPI в ответ говорили, что MPI поддерживает больше функций и, кроме того, эта система стандартизована, что подтверждается официальным документом. Соглашаясь с этим, защитники PVM возражали, что отсутствие всех бумаг, связанных с бюрократическим процессом стандартизации, — еще не повод отказываться от системы. В конечном итоге, когда все было сказано и сделано, победа досталась MPI.

Планирование

Программистам систем MPI несложно разрабатывать задания, в которых запросы делаются сразу к нескольким процессорам и которые выполняются достаточно долго. При наличии нескольких запросов от нескольких пользователей для обработки каждого запроса задействуется разное количество процессоров на разное время. Поэтому кластеру необходим планировщик, определяющий, когда запускать каждое из заданий.

В простейшей модели планировщику заданий достаточно, сколько процессоров требуется для выполнения каждого задания. В этой модели задания выстраиваются строго в порядке FIFO (рис. 8.40, *а*). Когда первое задание начинает выполняться, планировщик проверяет, достаточно ли процессоров для выполнения следующего по порядку задания. Если да, то оно тоже начинает выполняться и т. д. Если нет, то система ждет, пока не освободится достаточное количество процессоров. В нашем примере кластер содержит 8 процессоров, но он вполне мог бы содержать 128 в блоках по 16 штук (получилось бы 8 групп процессоров) или в какой-нибудь другой комбинации.

В более сложном алгоритме планирования удается избежать блокирования начала очереди. В этом случае все задания, для которых не хватает процессоров, пропускаются, а на выполнение ставится первое же задание, процессоров для которого достаточно. Всякий раз, когда завершается выполнение задания, очередь из оставшихся заданий проверяется в порядке FIFO (рис. 8.40, *б*).

Еще более сложный алгоритм требует, чтобы заранее было известно, сколько процессоров нужно для выполнения каждого задания и сколько времени на это потребуется. Располагая этой информацией, планировщик заданий может попытаться заполнить заданиями прямоугольник в системе координат процессоры — время (рис. 8.40, *в*). Это особенно эффективно, когда задания получены днем, а выполняться должны ночью. В этом случае планировщик заданий получает всю необходимую информацию заранее и может выполнять задания в оптимальном порядке.

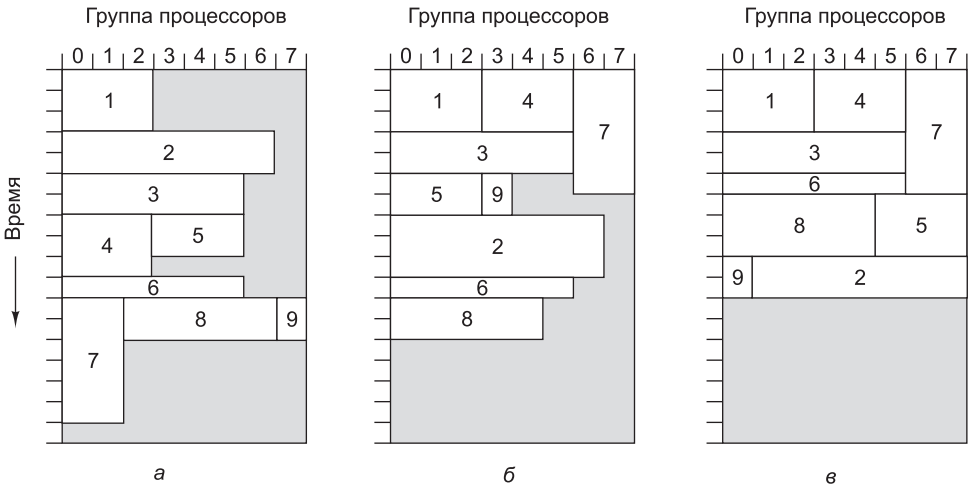


Рис. 8.40. Выполнение заданий в кластере (серым цветом показаны простаивающие процессоры): в порядке FIFO (*а*); без блокирования начала очереди (*б*); путем заполнения прямоугольника в системе координат процессоры — время (*в*)

Общая память на прикладном уровне

Из наших примеров видно, что мультикомпьютеры поддаются масштабированию гораздо лучше, чем мультипроцессоры. Этот факт привел к возникновению систем передачи сообщений, таких как MPI. Многим программистам эта модель не нравится, и они предпочли бы иметь иллюзию общей памяти, даже если ее на самом деле не существует. Если бы удалось достичь этой цели, это было бы прекрасно во всех отношениях: и в отношении удешевления аппаратуры (по крайней мере, на уровне каждого узла), и в отношении удобства программирования. Можно сказать, что это был бы Священный Грааль параллельных вычислений.

Многие исследователи пришли к выводу, что общую память не обязательно строить на архитектурном уровне — существуют другие пути. Из рис. 8.18 видно, что есть несколько уровней, на которых можно организовать общую память. Далее мы узнаем, как ввести общую память в программную модель мультикомпьютера, если аппаратно она не поддерживается.

Распределенная общая память

Один из классов систем с общей памятью на прикладном уровне — это системы со страничной организацией памяти. Этот класс систем известен под аббревиатурой **DSM** (Distributed Shared Memory — **распределенная общая память**). Идея проста: ряд процессоров в мультикомпьютере совместно используют общее виртуальное адресное пространство со страничной организацией. Самый простой вариант — каждая страница хранится в ОЗУ только одного процессора. На рис. 8.41, *а* мы видим общее виртуальное адресное пространство, которое состоит из 16 страниц, распределенных между четырьмя процессорами.

Когда процессор обращается к странице в своем локальном ОЗУ, чтение и запись происходят без задержки. Если же процессор обращается к странице другого ОЗУ, происходит ошибка отсутствия страницы. Однако вместо того чтобы искать отсутствующую страницу на диске, операционная система посылает сообщение

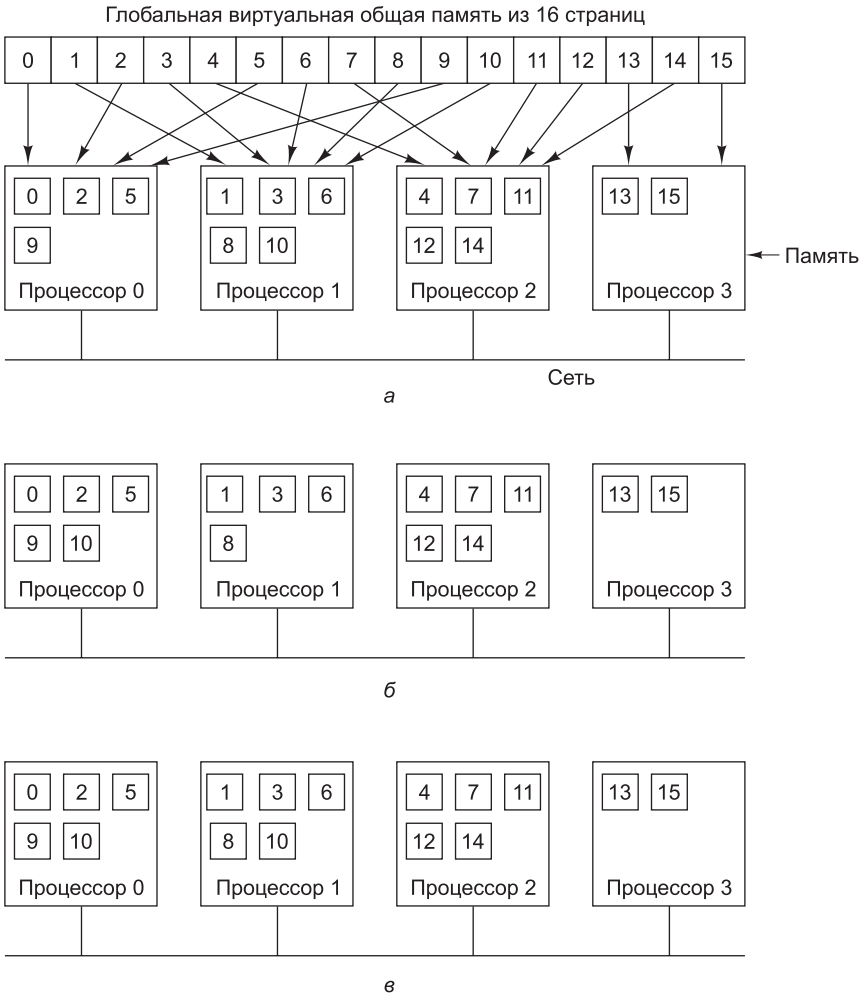


Рис. 8.41. Виртуальное адресное пространство из 16 страниц, распределенных между четырьмя узлами мультикомпьютера: исходное состояние (*а*); состояние после обращения процессора 0 к странице 10 (*б*); состояние после обращения процессора 1 к странице 10, предназначенной только для чтения (*в*)

в узел, в котором находится данная страница, чтобы извлечь ее из локального адресного пространства и отправить по назначению. После получения страницы она снова отображается на память, а приостановленная команда выполняется заново, как и при обычной ошибке отсутствия страницы. На рис. 8.41, *б* мы видим ситуацию после того, как процессор 0 получил ошибку отсутствия страницы 10, после чего та была передана из процессора 1 в процессор 0.

Впервые идея была реализована в машине IVY [Li and Hudak, 1989]. В ней мультикомпьютер обладает полнофункциональной секвенциально состоятельной общей памятью. В целях повышения производительности возможны разнообразные варианты оптимизации. Первая оптимизация в IVY — страницы, предназначенные только для чтения, могли присутствовать одновременно в нескольких

узлах. В случае ошибки отсутствия страницы в запрашивающую машину посылается копия этой страницы, но оригинал остается на месте, поскольку нет никакой опасности конфликтов. На рис. 8.41, *в* показана ситуация, когда два процессора совместно используют общую страницу 10, предназначенную только для чтения.

Но даже при такой оптимизации трудно достичь высокой производительности, особенно когда один процесс записывает несколько слов в верхней части какой-либо страницы, а другой процесс на другом процессоре в это же время записывает несколько слов в нижней части той же страницы. Поскольку разрешено существование только одной копии этой страницы, страница постоянно должна передаваться туда и обратно. Подобная ситуация называется **мнимым разделением**.

Проблему мнимого разделения можно решать по-разному. Например, в системе Treadmarks проектировщики отказались от секвенциальной состоятельности в пользу свободной состоятельности [Amza, 1996]. В случае свободной состоятельности страницы, которые потенциально пригодны для записи, могут одновременно присутствовать на нескольких узлах, но перед записью процесс должен совершить операцию `acquire`, чтобы сообщить о своем намерении. В этот момент все копии, кроме последней, объявляются недействительными, и до выполнения операции `release` никаких копий создавать нельзя. После выполнения операции `release` страница вновь становится общедоступной.

Второй способ оптимизации в Treadmarks предполагает, что изначально все страницы отображаются на память в режиме только чтения. Когда запись в страницу производится впервые, система создает копию страницы, называемую **двойником**. Затем исходная страница отображается на память в формате, пригодном для чтения и записи, после чего последующие записи могут производиться без задержек. Если на удаленном узле происходит ошибка отсутствия страницы, что означает необходимость передачи туда модифицированной страницы, сначала выполняется пословное сравнение текущей страницы с ее двойником. После этого на удаленный узел передаются только те слова, которые были модифицированы, а не вся страница, что сокращает размер сообщений.

После возникновения ошибки отсутствия страницы нужно определить, где ее искать. Здесь возможны разные подходы, в том числе использовать каталоги, как в NUMA- и COMA-машинах. Многие решения, применяемые в DSM, пригодны и для NUMA- и COMA-машин, поскольку DSM — это программная реализация таких машин, в которой каждая страница трактуется как строка кэша.

DSM по-прежнему остается ареной активных исследований. Большой интерес представляют системы CASHMERE [Kontothanassis, et al., 1997; Stets et al., 1997], CRL [Johnson et al., 1995], Shasta [Scales et al., 1996] и Treadmarks [Amza, 1996; Lu et al., 1997].

Linda

Системы DSM со страничной организацией памяти (такие как IVY и Treadmarks) используют диспетчера памяти, чтобы аппаратно перехватывать доступ к отсутствующим страницам. Хотя подготовка и пересылка только различающихся слов вместо всей страницы положительно сказывается на производительности, страницы остаются неудобными объектами совместного использования, поэтому применяются и другие подходы.

Один из таких подходов реализован в система Linda, в которой процессы на разных машинах получают в свое распоряжение высокоструктурированную распределенную общую память [Carriero and Gelernter,1989]. Доступ к этой памяти осуществляется с помощью минимального набора примитивов, которые можно включать в существующие языки (например, в C или FORTRAN), в результате формируются так называемые параллельные языки — в данном случае это C-Linda и FORTRAN-Linda.

В основе системы Linda лежит понятие абстрактного **пространства кортежей**, которое глобально по отношению ко всей системе и доступно всем процессам этой системы. Пространство кортежей похоже на глобальную общую память, только с определенной внутренней структурой. Каждый **кортеж** в пространстве кортежей состоит из одного или нескольких полей. В C-Linda поля могут содержать целые, длинные целые и числа с плавающей точкой, а также сложные типы данных, например массивы (в том числе символьные строки) и структуры (но не другие кортежи). В листинге 8.1 приведено три примера кортежей.

Листинг 8.1. Кортежи в Linda

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is sister", Carolyn, Elinor)
```

С кортежами можно выполнять 4 операции. Первая из них, `out`, вводит кортеж в пространство кортежей. Например:

```
out("abc", 2, 5);
```

Эта операция вводит кортеж `("abc", 2, 5)` в пространство кортежей. Поля операции `out` обычно содержат константы, переменные или выражения, например:

```
out("matrix-1", i, j, 3.14);
```

Эта операция вводит в пространство кортежей кортеж с четырьмя полями, причем второе и третье поля определяются по текущим значениям переменных `i` и `j`.

Выборка кортежа из пространства кортежей осуществляется примитивом `in`. Обращение к кортежам производится по содержанию, а не по имени или адресу. Поля операции `in` могут содержать выражения или формальные параметры. Например:

```
in("abc", 2, ? i);
```

Эта операция «ищет» в пространстве кортежей кортеж, состоящий из символьной строки `"abc"`, целого числа 2 и третьего поля, которое может содержать любое целое число (предполагается, что `i` — целое). Найденный кортеж извлекается из пространства кортежей, а переменной `i` присваивается значение третьего поля. Поиск совпадения и извлечение являются атомарными действиями. Поэтому когда два процесса одновременно выполняют одну и ту же операцию `in`, а в пространстве кортежей более нет совпадающих кортежей, успешной оказывается только одна из операций. Пространство кортежей может содержать несколько копий одного кортежа.

Алгоритм поиска совпадения, который используется в операции `in`, довольно прост. Поля операции `in`, называемые **шаблоном**, сравниваются с соответствую-

щими полями каждого кортежа в пространстве кортежей. Совпадение имеет место, если удовлетворены следующие три условия:

- шаблон и кортеж имеют одинаковое количество полей;
- типы соответствующих полей совпадают;
- каждая константа или переменная в шаблоне совпадает с соответствующим полем в кортеже.

Формальные параметры, представленные вопросительным знаком, за которым следует имя или тип переменной, в поиске совпадения не участвуют (за исключением проверки типа), хотя тем параметрам, которые содержат имя переменной, после нахождения совпадения присваиваются значения.

Если совпадающий кортеж не обнаруживается, процесс приостанавливается, пока другой процесс не введет нужный кортеж в пространство кортежей. В этот момент вызывающий процесс автоматически возобновляет работу и обнаруживает новый кортеж. Процессы блокируются и деблокируются автоматически, поэтому если один процесс готов выводить кортеж, а другой — вводить, то не имеет никакого значения, какой из процессов будет первым.

Третья операция, `read`, напоминает `in`, только она не удаляет кортеж из пространства кортежей. Четвертая операция, `eval`, на основе своих параметров строит новый кортеж, который копируется в пространство кортежей. Этот механизм можно использовать для выполнения любых вычислений. Именно так создаются параллельные процессы в системе Linda.

Основной программной парадигмой в Linda является **модель тиражируемых рабочих** (replicated worker model). В основе модели лежит понятие **пакета заданий** (task bag), которые должны быть выполнены. Основной процесс начинается с выполнения цикла, содержащего операцию

```
out("task-bag", job);
```

При каждом прохождении цикла одно из заданий вводится в пространство кортежей. Каждый рабочий процесс начинает функционирование с получения кортежа, описывающего задание:

```
in("task-bag", ?job);
```

Затем процесс выполняет это задание. По выполнении задания он получает следующее. Кроме того, во время выполнения в пакет заданий может быть включено новое задание. Таким образом, работа динамически распределяется среди рабочих процессов, и каждый рабочий процесс постоянно занят.

Существуют различные реализации Linda в мультикомпьютерных системах. Во всех реализациях ключевым является вопрос о том, как распределить кортежи по машинам и как их при необходимости находить. Возможные варианты — широковещание и каталоги. Репликация — это тоже важный вопрос. Подробнее об это см. [Bjornson, 1993].

Orca

Немного иной подход к реализации в мультикомпьютере общей памяти на прикладном уровне — задействовать для совместного использования не кортежи, а полноценные объекты. Как известно, объекты состоят из внутреннего (скрытого) состояния и методов для операций с этим состоянием. Поскольку программист

не имеет прямого доступа к состоянию, появляется возможность совместного использования ресурсов машинами, которые не имеют общей физической памяти.

Одна система, в которой реализован этот подход, называется Orca [Bal, 1991; Bal et al., 1992; Bal and Tanenbaum, 1988]. Orca — это традиционный язык программирования (в его основе лежит язык Modula 2), обладающий двумя новыми свойствами — поддержкой объектов и возможностью создавать новые процессы. Объект в Orca — это абстрактный тип данных, аналогичный объекту в Java или пакету в Ada. Объект инкапсулирует внутренние структуры данных и пользовательские методы, которые называются **операциями**. Объекты пассивны, то есть они не содержат программных потоков, которым можно посылать сообщения. Процессы получают доступ к внутренним данным объекта путем вызова его методов.

Каждый метод в Orca состоит из списка пар «предохранитель-блок операторов». Предохранитель — это логическое выражение, которое может принимать значение истина (`true`) или ложь (`false`). Когда вызывается операция, все ее предохранители оцениваются в произвольном порядке. Если все они ложные, вызывающий процесс приостанавливается до тех пор, пока один из них не станет истинным. При нахождении такого предохранителя выполняется следующий за ним блок операторов. В листинге 8.2 показан объект `stack` с двумя операциями, `push` и `pop`.

Листинг 8.2. Упрощенный объект stack с внутренними данными и двумя операциями

```
Object implementation stack;
  top:integer;                    # хранилище для стека
  stack:array[integer 0..N-1]of integer;

  operation push(item: integer); # функция, которая ничего не возвращает
  begin
    stack[top] := item;           # помещаем элемент в стек
    top := top + 1;               # увеличение указателя стека
  end;

  operation pop(): integer;       # функция, которая возвращает целое
  begin
    guard top>0 do                # приостанавливает работу, если стек пуст
      top := top – 1;             # уменьшает указатель стека
      return stack[top];          # возвращает вершину стека
    od;
  end;

begin
  top:=0;                         # инициализация
end;
```

После определения объекта `stack` можно объявлять переменные этого типа:

```
s, t, stack;
```

Такая запись создает два объекта стека и устанавливает переменную `top` в каждом объекте на 0. Целочисленную переменную `k` можно поместить в стек `s` с помощью оператора

```
s$push(k);
```

Операция pop содержит предохранитель, поэтому попытка вытолкнуть переменную из пустого стека вызовет приостановку вызывающего процесса до тех пор, пока другой процесс не поместит что-либо в стек.

В Orca имеется оператор **порождения нового процесса** fork. Новый процесс запускает процедуру, указанную в операторе fork. Новому процессу могут передаваться и другие параметры, в том числе объект. Именно так объекты распределяются по машинам. Например:

```
for i in 1..n do fork foobar(s) on i; od;
```

Этот оператор порождает новый процесс на всех машинах с 1 по n, запуская программу foobar в каждом из них. Поскольку эти n новых процессов (а также исходный процесс) работают параллельно, они все могут помещать элементы в общий стек s и извлекать элементы из общего стека s, как будто все они работают на мультипроцессоре с общей памятью. Исполнительная система создает иллюзию общей памяти, которой в действительности не существует.

Операции с совместно используемыми объектами атомарны и секвенциально состоятельны. То есть система гарантирует, что если несколько процессов выполняют операции с одним объектом практически одновременно, система сама устанавливает определенную очередность событий, которую воспринимают все процессы.

В Orca общие данные и синхронизация реализованы иначе, чем в системах со страничной организацией памяти. В параллельных программах требуются два вида синхронизации. Первый — взаимное исключение. Благодаря взаимному исключению два процесса не могут одновременно выполнять одну и ту же критическую секцию кода. В Orca каждая операция с совместно используемым объектом напоминает выполнение критической секции, поскольку система гарантирует, что конечный результат будет таким же, как если бы все критические секции кода выполнялись поочередно. В этом отношении объект в Orca похож на распределенный монитор [Hoare, 1975].

Второй вид синхронизации — условная синхронизация, при которой процесс блокируется, ожидая выполнения определенного условия. В Orca условная синхронизация осуществляется при помощи предохранителей. В примере, приведенном в листинге 8.2, процесс, который пытается извлечь из пустого стека элемент, блокируется до появления элементов в стеке.

Orca поддерживает репликацию, миграцию и состоятельность объектов, а также вызов для них операций. Каждый объект может находиться в одном из двух состояний: он может быть единственным или реплицированным. В первом случае объект существует только на одной машине, поэтому все запросы направляются туда. Реплицированный объект присутствует на всех машинах, на которых запущен процесс, использующий этот объект. Это упрощает операцию чтения (поскольку ее можно производить локально), но усложняет обновление. При выполнении операции, которая изменяет реплицированный объект, сначала нужно получить от центрального процесса порядковый номер процесса. Затем в каждую машину, содержащую копию объекта, отправляется сообщение о необходимости выполнить указанную операцию. Поскольку все такие обновления обладают порядковыми номерами, каждая машина просто выполняют операции в указанном порядке, что гарантирует секвенциальную состоятельность.

Производительность

Цель создания параллельного компьютера — добиться, чтобы он работал быстрее, чем однопроцессорная машина. Если эта цель не достигнута, никакого смысла в разработке параллельного компьютера нет. Более того, эта цель должна быть достигнута при минимальных затратах. Машина, которая работает в два раза быстрее, чем однопроцессорная, но стоит в 50 раз дороже последней, вряд ли будет пользоваться спросом. В этом подразделе мы рассмотрим некоторые аспекты производительности параллельных компьютерных архитектур.

Аппаратные метрики

С позиции аппаратуры наибольший интерес представляет быстродействие процессоров, устройств ввода-вывода и коммуникационной сети. Поскольку скорость работы процессоров и устройств ввода-вывода такая же, как и в однопроцессорной машине, ключевыми в параллельной системе являются параметры коммуникационной сети. Здесь есть две ключевых метрики: время запаздывания и пропускная способность. Мы рассмотрим их по очереди.

Полное время запаздывания, или время оборота, — это время, которое требуется на то, чтобы процессор отправил пакет и получил ответ. Если пакет посылается в память, то время запаздывания — это время, которое требуется на чтение и запись слова или блока слов. Если пакет посылается другому процессору, то время запаздывания — это время, которое требуется на передачу пакетов данного размера между процессорами. Обычно интерес представляет время запаздывания для пакетов минимального размера (как правило, одного слова или небольшой строки кэша).

Величину времени запаздывания определяют несколько факторов, и это время разное для технологий коммутации каналов, коммутации с сохранением и продвижением, виртуальной сквозной маршрутизации. В случае коммутации каналов время запаздывания составляет сумму времени установки соединения и времени передачи. Для установки соединения высылается пробный пакет, позволяющий зарезервировать необходимые ресурсы, обратно возвращается сообщение с отчетом. После этого можно компонововать пакет данных. Когда пакет готов, биты можно передавать на полной скорости, поэтому если общее время установки соединения составляет T_s, размер пакета равен p бит, а пропускная способность — b бит/с, время запаздывания в одну сторону составляет $T_s + p/b$. Если схема дуплексная и для ответа установки соединения не требуется, минимальное время запаздывания при передаче пакета размером в p бит и получения ответа размером в p бит составляет $T_s + 2p/b$ секунд.

При коммутации пакетов посылать получателю пробный пакет заранее не нужно, но все равно требуется некоторое время T_a на компоновку пакета. Здесь время передачи в одну сторону составляет $T_a + p/b$, но за этот период пакет доходит только до первого коммутатора. При прохождении через сам коммутатор получается некоторая задержка, T_d, затем происходит переход к следующему коммутатору и т. д. Время T_d состоит из времени обработки и задержки в очереди (когда нужно ждать, пока освободится выходной порт). Если имеется n коммутаторов, то общее время запаздывания в одну сторону составляет $T_a + n(p/b + T_d) + p/b$, где последнее слагаемое отражает факт передачи пакета от последнего коммутатора к получателю.

Время запаздывания в одну сторону для виртуальной сквозной маршрутизации в лучшем случае приближается к значению $T_a + p/b$, поскольку здесь нет пробных пакетов для установки соединения и нет задержки, обусловленной промежуточным хранением. По существу, это — время компоновки пакета плюс время передачи битов. Следовало бы еще прибавить задержку на распространение сигнала, но она во всех случаях невелика.

Следующая аппаратная метрика — пропускная способность. Многие параллельные программы, особенно в естественных науках, ориентированы на перемещение огромных объемов данных, поэтому число байтов, которое система способна передавать в секунду, является очень важным показателем производительности. Существуют несколько метрик пропускной способности. Одну из них — пропускную способность сечения — мы уже рассмотрели (см. подраздел «Коммуникационные сети» в разделе «Мультикомпьютеры»). Другая метрика — **совокупная пропускная способность** — вычисляется путем суммирования пропускной способности всех линий связи. Это число показывает максимальное количество битов, которое можно передать единовременно. Еще одна важная метрика — средняя пропускная способность каждого процессора. Если каждый процессор способен производить данные только со скоростью 1 Мбайт/с, то от сети с пропускной способностью сечения в 100 Гбайт/с проку мало. Скорость взаимодействия в этом случае ограничивается скоростными возможностями каждого процессора.

На практике приблизиться к теоретически возможной пропускной способности очень трудно. Причины этого могут быть самыми разными. Например, каждый пакет всегда содержит служебные данные, относящиеся к компоновке, созданию заголовка, отправке. При отправке 1024 пакетов по 4 байта каждый мы никогда не достигнем той же пропускной способности, что и при отправке одного пакета на 4096 байт. Однако для сокращения времени запаздывания лучше использовать маленькие пакеты, поскольку большие надолго блокируют линии и коммутаторы. В результате имеет место конфликт между способами достижения низкого времени запаздывания и высокой пропускной способности. Для одних прикладных задач важнее время запаздывания, для других — пропускная способность. Но в любом случае важно понимать, что пропускную способность всегда можно увеличить за счет дополнительных материальных затрат (добавив больше проводов или установив более широкие провода), а вот что касается сокращения времени запаздывания, финансовые вливания здесь не помогут. Поэтому обычно лучше с самого начала позаботиться о минимальном времени запаздывания, а уже потом думать о пропускной способности.

Программные метрики

Аппаратные метрики, такие как время запаздывания и пропускная способность, показывают, на что способно аппаратное обеспечение. Но пользователей интересует совсем другое. Они хотят знать, насколько быстрее будут работать их программы на параллельном компьютере по сравнению с однопроцессорным. Для них ключевой метрикой является ускорение: насколько быстрее работает программа в n-процессорной системе по сравнению с однопроцессорной. Результаты обычно иллюстрируются графически (рис. 8.42). Здесь мы видим несколько разных параллельных программ, которые работают на мультикомпьютере, состоя-

щем из 64 процессоров Pentium Pro. Каждая кривая показывает ускорение одной программы с k процессорами как функцию от k. Пунктирной линией обозначено идеальное ускорение, при котором использование k процессоров заставляет программу работать в k раз быстрее для любого k. Лишь немногие программы достигают идеального ускорения, хотя существует довольно много программ, которые приближаются к идеалу. Проблема моделирования N тел за счет параллелизма решается гораздо быстрее, авари (африканская игра, также называемая «калах») тоже обсчитывается быстрее, а вот ускорить инвертирование заданной профильной матрицы более чем в пять раз нельзя, сколько бы процессоров мы ни использовали. Программы и результаты обсуждаются в [Bal et al., 1998].

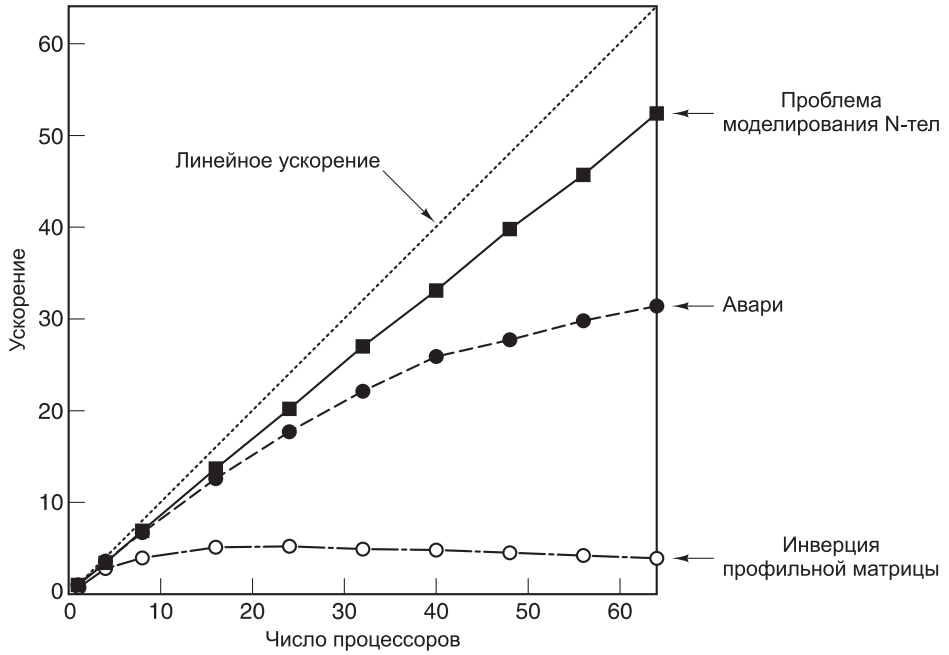


Рис. 8.42. На практике программы не могут достичь идеального ускорения (показано пунктирной линией)

Есть ряд причин, по которым практически невозможно достичь идеального ускорения: практически во всех программах есть фрагменты, принципиально выполняемые последовательно, например инициализация, считывание исходных данных или получение результатов. Увеличение числа процессоров здесь не поможет. Предположим, что на однопроцессорном компьютере программа работает T секунд, причем доля (f) от этого времени выполняется последовательно, а доля $(1 - f)$ потенциально может выполняться параллельно, как показано на рис. 8.43, *а*. Если параллельный код можно запустить на n процессорах, то время выполнения этого кода в лучшем случае сократится с $(1 - f)T$ до $(1 - f)T/n$, как показано на рис. 8.43, *б*. В результате общее время выполнения программы (и последовательного, и параллельного кода) составит $fT + (1 - f)T/n$. Ускорение — это время выполнения исходной программы (T), разделенное на это новое время:

$$\text{Ускорение} = \frac{n}{1+(n-1)f}.$$

Для $f = 0$ мы можем получить линейное ускорение, но для $f > 0$ идеальное ускорение недостижимо, поскольку в программе имеется последовательная часть. Это явление носит название **закона Амдала**.

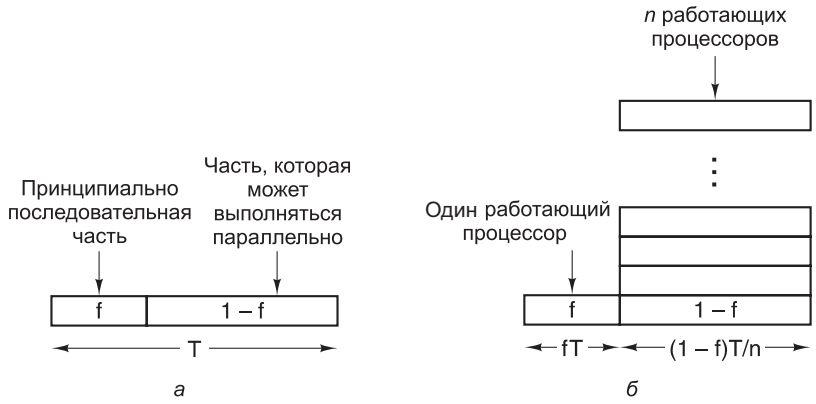


Рис. 8.43. Помимо последовательной части программа содержит ту часть, которая может выполняться параллельно (*а*); результат параллельной обработки части программы (*б*)

Закон Амдала — только одна из причин, по которой идеальное ускорение недостижимо. Определенную роль в этом играют и время запаздывания в линиях связи, и ограниченная пропускная способность, и недостатки алгоритмов. Даже если бы у нас было 1000 процессоров, не все программы можно написать так, чтобы все их использовать, а издержки, связанные с запуском стольких процессоров, могут быть весьма значительными. Больше того, многие известные алгоритмы почти не поддаются параллельной обработке, поэтому их приходится заменять квазиоптимальными алгоритмами. В то же время для многих прикладных задач весьма желательно было бы заставить программу работать в n раз быстрее, даже если для этого потребуются $2n$ процессоров. В конце концов, процессоры не такие уж и дорогие, а многие компании не расстраиваются из-за того, что в других областях их бизнеса эффективность существенно ниже 100 %.

Приемы повышения производительности

Самый очевидный способ поднять производительность системы — добавить процессоры. Однако добавлять процессоры нужно таким образом, чтобы в системе не появлялись узкие места. Система, после добавления процессоров в которую имеет место соответствующий прирост производительности, называется **масштабируемой**.

Рассмотрим 4 процессора, связанные общей шиной (рис. 8.44, *а*). Представьте, что мы расширили систему до 16 процессоров, добавив еще 12 (рис. 8.44, *б*). Если пропускная способность шины составляет b Мбайт/с, то увеличив в 4 раза число процессоров, мы сократим доступную каждому процессору пропускную способность с $b/4$ до $b/16$ Мбайт/с. Такую систему нельзя назвать масштабируемой.

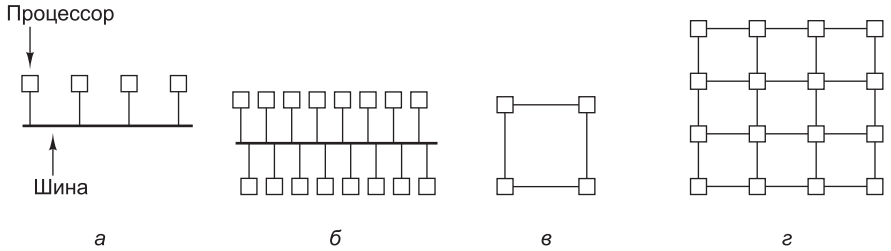


Рис. 8.44. Система из 4 процессоров, связанных общей шиной (*а*); система из 16 процессоров, связанных общей шиной (*б*); коммуникационная решетка из 4 процессоров (*в*); коммуникационная решетка из 16 процессоров (*г*)

А теперь проделаем то же самое с коммуникационной решеткой (рис. 8.44, *в*, *г*). В такой топологии добавление новых процессоров означает появление новых линий связи, поэтому при масштабировании системы совокупная пропускная способность каждого процессора не снижается, как в случае с шиной. Фактически, отношение числа линий связи к числу процессоров увеличивается от 1,0 при наличии 4 процессоров (4 линии связи) до 1,5 при наличии 16 процессоров (24 линии связи), поэтому с добавлением новых процессоров совокупная пропускная способность каждого процессора растет.

Естественно, пропускная способность — не единственный параметр. Добавление процессоров к шине не увеличивает диаметр сети или время запаздывания, в то время как добавление процессоров к решетке, напротив, увеличивает. Диаметр решетки размером $n \times n$ равен $2(n - 1)$, поэтому в худшем случае время запаздывания растет примерно как квадратный корень от числа процессоров. Для 400 процессоров диаметр равен 38, для 1600 процессоров — 78, поэтому если увеличить число процессоров в 4 раза, то диаметр и, следовательно, среднее время запаздывания вырастут приблизительно вдвое.

В идеале масштабируемая система при добавлении новых процессоров должна сохранять одну и ту же среднюю пропускную способность в расчете на один процессор и постоянное среднее время запаздывания. На практике сохранение достаточной пропускной способности каждого процессора осуществимо, но время запаздывания с увеличением числа процессоров растет. Лучше всего было бы сделать так, чтобы она росла логарифмически, как в гиперкубе.

Дело в том, что время запаздывания часто является фатальным в отношении производительности мелкомодульных и среднемодульных приложений. Если программе требуются данные, которых нет в ее локальной памяти, на их получение требуется существенное время, причем чем больше система, тем больше получается задержка. Эта проблема характерна и для мультипроцессоров, и для мультикомпьютеров, поскольку в обеих архитектурах физическая память разделена на фиксированные распределенные модули.

Системные разработчики применяют несколько технологий, позволяющих сократить или, по крайней мере, замаскировать время запаздывания. Первая технология — это репликация данных. Если копии блока данных можно хранить в нескольких местах, то скорость доступа к этим данным можно повысить. Один из возможных вариантов — использование кэша, когда одна или несколько копий блоков данных хранятся рядом с местом, в котором они могут понадобиться и которому они «принадлежат». Другой вариант — хранение нескольких

равноправных копий (в противоположность неравноправным отношениям главная-подчиненная, характерным для основной памяти и кэш-памяти). В этом случае очень важно, кем, когда и где эти копии размещаются. Здесь возможны самые разные варианты, от аппаратного динамического размещения данных по запросу до их принудительного размещения во время загрузки с помощью соответствующих директив компилятора. В любом случае на первый план выходит задача управления согласованием.

Вторая технология — маскирование времени запаздывания путем так называемой **упреждающей выборки** (prefetching), при которой элемент данных вызывается еще до того, как он понадобится. Это позволяет перекрыть процесс вызова и процесс выполнения, так как программа получает затребованный элемент данных без задержки. Упреждающая выборка может быть реализована как аппаратно, так и программно. В случае упреждающей выборки в кэш загружается не только нужное слово, но и вся содержащая его строка кэша в расчете на то, что другие слова из этой строки пригодятся в будущем.

Упреждающей выборкой можно управлять и непосредственно. Когда компилятор выясняет, что программе в ходе выполнения потребуются те или иные данные, он вставляет в код команду их получения, причем с таким расчетом, чтобы получить нужные данные вовремя. Такая стратегия требует, чтобы компилятор обладал полными знаниями о машине и механизме ее синхронизации, а также контролировал место хранения всех данных. Спекулятивные команды `LOAD` работают лучше всего, когда абсолютно точно известно, что загруженные данные будут использованы. Ошибка отсутствия страницы при выполнении команды `LOAD` для ветви, которая никогда не понадобится, — дело чрезвычайно затратное.

Третья технология маскирования времени запаздывания — использование нескольких программных потоков. Если переключение между процессами может выполняться достаточно быстро (например, за счет выделения каждым из них собственной карты памяти и собственных регистров), то когда один поток блокируется в ожидании данных, аппаратно можно быстро переключиться на другой поток, готовый к выполнению. В предельном случае процессор выполняет первую команду из потока 1, вторую команду из потока 2 и т. д. Таким образом, процессор всегда занят, даже при длительных задержках отдельных потоков.

Четвертая технология маскирования времени запаздывания — использование неблокирующих операций записи. Обычно при выполнении команды `STORE` процессор ждет ее завершения и только после этого продолжает работу. При наличии неблокирующих записей программа продолжает работу, даже когда выполняется операция с памятью. Продолжать работу программы при выполнении команды `LOAD` сложнее, но даже это возможно за счет исполнения с изменением последовательности.

Распределенные вычисления

Многие сегодняшние задачи в науке, технике, производстве, в области охраны окружающей среды и в других областях очень масштабны и многоплановы. Для их решения требуется объединить знания, умения, средства, возможности, программы и данные многих организаций, зачастую разбросанных по всему миру.

Вот лишь некоторые примеры:

- ученые, исследующие все аспекты миссии на Марс;
- консорциум, разрабатывающий сложный продукт (например, самолет или дамбу);
- интернациональная спасательная команда, координирующая свои действия во время стихийного бедствия.

Для некоторых задач требуется более долговременное сотрудничество, для других менее, но у всех есть общая деталь. Разные организации вместе работают для достижения общей цели, используя каждая собственные ресурсы и собственные производственные процедуры.

До недавнего времени было очень сложно обеспечить совместную работу разных организаций, в которых используются разные операционные системы, разные базы данных и протоколы. Однако рост потребности в крупномасштабном сотрудничестве между организациями привел к развитию систем и технологий объединения разрозненных компьютеров в то, что получило название **распределенных вычислений** (grid computing). В определенном смысле, распределенные вычисления — это следующий шаг вдоль оси на рис. 8.1. Систему распределенных вычислений можно рассматривать как очень большой, интернациональный слабо связанный гетерогенный кластер.

Целью системы распределенных вычислений является создание технической инфраструктуры, которая позволила бы из нескольких организаций, преследующих общую цель, создать единую **виртуальную организацию**. Эта виртуальная организация должна быть гибкой структурой с динамически изменяющимся числом членов, в которой отдельные члены могут работать вместе в требуемых областях и в то же время полностью контролировать собственные ресурсы. Для этой цели исследователями разрабатываются службы, инструменты и протоколы, и именно это позволяет отдельным членам функционировать в рамках виртуальной организации.

Система распределенных вычислений по своей сути многомерна, с большим количеством участников — одноранговых узлов. Ее можно противопоставить традиционным моделям вычислений. В модели клиент-сервер в транзакцию вовлечены два участника — сервер, который предоставляет некоторую услугу, и клиент, желающий ее получить. Типичным примером является Всемирная паутина, в которой множество пользователей обращается к серверам за информацией. Отличаются системы распределенных вычислений и от двухточечных приложений, объединяющих пары машин для обмена файлами друг с другом. Типичным примером двухточечного приложения является электронная почта. Как следствие этих отличий, необходимы новые протоколы и новые технологии.

В системе распределенных вычислений необходимо обеспечить доступ к самым разным ресурсам. У каждого ресурса есть своя система и владеющая им организация, которая решает, какая часть ресурса доступна, в какое время и кому. Если не вдаваться в детали, можно сказать, что суть системы распределенных вычислений в управлении доступом к ресурсам.

Один из вариантов моделирования системы распределенных вычислений заключается в ее представлении в виде многоуровневой иерархической структуры (табл. 8.7). Нижний уровень — это **уровень инфраструктуры**, объединяющий компоненты, из которых построена система распределенных вычислений. В части

аппаратного обеспечения сюда входят процессоры, диски, сети и сенсоры, в части программного — программы и данные. Это те физические ресурсы, доступ к которым поддерживается системой распределенных вычислений.

Таблица 8.7. Уровни иерархии системы распределенных вычислений

Уровень	Описание
Уровень приложений	Приложения, которые совместно и согласованно используют ресурсы
Уровень коллективов	Исследования, посредничество, мониторинг, управление группами ресурсов
Уровень ресурсов	Безопасность и управление доступом к отдельным ресурсам
Уровень инфраструктуры	Физические ресурсы, включая компьютеры, дисковую память, сети, сенсоры, программы и данные

Следующий уровень вверх по иерархии — **уровень ресурсов**. Этот уровень отвечает за управление отдельными ресурсами. Зачастую с включенным в систему распределенных вычислений ресурсом связан локальный процесс, который управляет ресурсом и обеспечивает контролируемый доступ к нему удаленных пользователей. Назначение этого уровня состоит в том, чтобы предоставить более высоким уровням единообразный интерфейс, при помощи которого они могли бы выяснять характеристики отдельных ресурсов, выполнять их мониторинг и безопасно использовать.

Еще выше лежит **уровень коллективов**, оперирующий группами ресурсов. Одной из его функций является поиск ресурсов, посредством которого пользователи находят необходимые им такты процессора, дисковое пространство или конкретные данные. Для предоставления необходимой информации уровень коллективов может поддерживать каталоги и другие базы данных. Кроме того, он может выполнять посреднические операции, сводя поставщиков и пользователей различных услуг, а также, возможно, распределяя дефицитные ресурсы между конкурирующими пользователями. Уровень коллективов отвечает также за размножение данных, включение в систему распределенных вычислений новых участников и ресурсов, учет и поддержание баз данных политик доступа, описывающих, какому пользователю какие ресурсы доступны.

На вершине иерархии находится **уровень приложений**. На этом уровне работают пользовательские приложения. Уровень приложений обращается к нижележащим уровням, чтобы получать права на использование тех или иных ресурсов, отправлять запросы на их использование, отслеживать ход выполнения запросов, обрабатывать отказы, извещать пользователя о результатах.

Ключевым фактором успеха системы распределенных вычислений является безопасность. Владельцы почти всегда настаивают на праве полного контроля над своими ресурсами с подробным мониторингом (кто, как долго и сколько их использует). Без хорошей системы безопасности ни одна организация не предложила бы свои ресурсы для распределенных вычислений. Однако если бы пользователю приходилось вводить свои имя и пароль на каждом нужном ему компьютере, вовлеченном в распределенные вычисления, работа пользователя стала бы слишком обременительной. Следовательно, должна быть выработана модель безопасности, учитывающая эти соображения.

Возможность однократной регистрации в системе — одна из ключевых характеристик модели безопасности. Использование системы распределенных вычислений начинается с регистрации и получения сертификата, то есть документа с цифровой подписью, указывающего, в чьих интересах должна выполняться работа. Сертификат может быть делегирован, так что если в процессе вычислений потребуются некоторые вспомогательные вычисления, дочерние процессы тоже могут быть идентифицированы с его помощью. Когда сертификат доступа представляется удаленной системе, он должен быть отображен на ее локальный механизм защиты. Например, в UNIX пользователи идентифицируются 16-разрядными идентификаторами, но в других системах могут применяться иные схемы. Наконец, необходим механизм, при помощи которого будут устанавливаться, поддерживаться и обновляться политики доступа.

Для поддержания взаимодействия между различными организациями и машинами необходимы стандарты как на предоставляемые услуги, так и на протоколы доступа к ним. Для управления процессом стандартизации сообщество распределенных вычислений создало организацию под названием Global Grid Forum. Результатом ее работы стал шаблон для формирования и развития различных стандартов, названный **OGSA** (Open Grid Services Architecture — **открытая архитектура служб распределенных вычислений**). Разрабатываемые стандарты по возможности опираются на существующие, например, для описания служб OGSA применяется язык WSDL (Web Services Definition Language — язык описания веб-служб). Стандартизованные на настоящее время службы попадают в одну из восьми категорий, хотя этот список, несомненно, в будущем будет расширен.

1. Службы инфраструктуры (обеспечивают взаимодействие между ресурсами).
2. Службы управления ресурсами (резервирование и освобождение ресурсов).
3. Службы данных (копирование и перемещение данных туда, где они нужны).
4. Контекстные службы (описание требуемых ресурсов и политик их использования).
5. Информационные службы (получение информации о доступности ресурса).
6. Службы самоконтроля (поддержание заявленного качества услуги).
7. Службы защиты (применение политик безопасности).
8. Службы управления выполнением (управление потоком задач).

О системах распределенных вычислений можно говорить еще очень много, но объем книги не позволяет нам дальше углубляться в эту тему. Дополнительную информацию об этих системах можно найти в [Abramson, 2011; Balasangameshwara and Raju, 2012; Celaya and Arronategui, 2011; Foster and Kesselman, 2003; Lee et al., 2011].

Краткое содержание главы

Добиваться повышения производительности компьютеров просто за счет увеличения тактовой частоты становится все сложнее, так как появляется проблема отвода тепла. Поэтому разработчики обратили свое внимание на параллелизм как на средство ускорения вычислений. Параллелизм может вводиться на раз-

ных уровнях, как на самых нижних, где элементы очень жестко связаны друг с другом, так и на верхних, где связи весьма слабые.

Нижний уровень — это внутрипроцессорный параллелизм, когда параллельные действия выполняются на базе единственной микросхемы. Одной из форм внутрипроцессорного параллелизма является параллелизм на уровне команд; в этом случае одна команда или последовательность команд разбивается на микрооперации, которые могут выполняться параллельно разными функциональными блоками. Второй формой внутрипроцессорного параллелизма является многопоточность; в этом случае процессор может поочередно переключаться между несколькими программными потоками. В результате получается виртуальный мультипроцессор. Третьей формой внутрипроцессорного параллелизма является однокристальный мультипроцессор. В однокристальном мультипроцессоре два или более ядер размещаются на одной микросхеме, что позволяет им работать одновременно.

На следующем уровне вверх по иерархии располагаются сопроцессоры. Обычно сопроцессор выполняется в виде встраиваемой платы, которая позволяет увеличить вычислительные возможности процессора в некоторых специальных областях, таких как обслуживание сетевого протокола или обработка мультимедийных данных. Эти дополнительные процессоры снижают нагрузку на главный процессор, предоставляя ему возможность делать другие вещи, пока они решают узкоспециализированные задачи.

Уровнем выше находятся мультипроцессоры с общей памятью. Такие системы содержат два или более полноценных процессора, совместно использующих общую память. Мультипроцессоры с однородным доступом к памяти (UMA-машины) могут взаимодействовать через общую шину (шину слежения) либо через сеть с перекрестной или многоступенчатой коммутацией. Они характеризуются унифицированным временем доступа к любым модулям памяти. В противоположность им в NUMA-мультипроцессорах всем процессам также предоставляется общее адресное пространство, но время доступа к удаленным модулям заметно больше, чем к локальным. Наконец, в COMA-мультипроцессорах строки кэша по запросу перемещаются от машине к машине, но «настоящего дома», как в других схемах, у этих строк нет.

Мультикомпьютерами называют системы с множеством процессоров, но без общей памяти. Каждый из них имеет собственную локальную память и связывается с другими посредством сообщений. MPP-машины, такие как BlueGene/P производства IBM, — это большие мультикомпьютеры, связанные специализированными коммуникационными сетями. Кластеры представляют собой более простые системы с доступными компонентами (как, например, ядро, обеспечивающее работу Google).

Мультикомпьютеры часто программируются с помощью специальных программных пакетов, таких как MPI, которые позволяют разрабатывать приложения, ориентированные на передачу сообщений. Альтернативные схемы связаны с использованием общей памяти на прикладном уровне. Сюда можно отнести разбитое на страницы адресное пространство в DSM, пространство кортежей в Linda, объекты в Orca и Globe. В DSM моделируется общая память на уровне страниц, и в этом система DSM напоминает NUMA-машину, исключая, возможно, более высокие издержки при удаленных обращениях.

На самом верхнем уровне располагаются системы распределенных вычислений. Это — наиболее слабо связанные системы; в них для решения общих задач объединяются целые организации, совместно используя через Интернет вычислительные возможности своих компьютеров, данные и другие ресурсы.

Вопросы и задания

1. Длина команды Intel x86 может достигать 17 байт. Является ли x86 VLIW-процессором?
2. Когда развитие технологии позволило размещать больше транзисторов на каждой микросхеме, Intel и AMD решили увеличить количество ядер на своих микросхемах. Могли ли они пойти по другому пути?
3. Пусть диапазон машинного представления числа составляет 0–255. Что получится в результате усечения чисел 96, –9, 300 и 256?
4. Совместимы ли следующие TriMedia-команды, и если нет, то почему?
   1) целочисленное сложение, целочисленное вычитание, загрузка, сложение с плавающей точкой, непосредственная загрузка;
   2) целочисленное вычитание, целочисленное умножение, непосредственная загрузка, сдвиг;
   3) непосредственная загрузка, сложение с плавающей точкой, умножение с плавающей точкой, ветвление, непосредственная загрузка.
5. На рис. 8.5, *г–д* показаны первые 12 циклов машинных команд. Для каждого варианта опишите, что происходит в следующих трех циклах.
6. На конкретном процессоре при кэш-промахе на уровне 1 и кэш-попадании на уровне 2 команда выполняется k машинных циклов. Пусть для маскировки кэш-промахов в кэше уровня 1 используется мелкомодульная многопоточность. Сколько программных потоков должны работать одновременно, чтобы полностью избежать холостых циклов?
7. Графический процессор NVIDIA Fermi напоминает одну из архитектур, описанных в главе 2. Какую именно?
8. Утром пчелиная матка созывает рабочих пчел и сообщает им, что сегодня нужно собрать нектар ноготков. Рабочие пчелы вылетают из улья и летят в разных направлениях в поисках ноготков. Что это за система, SIMD или MIMD?
9. Обсуждая модели состоятельности памяти, мы упомянули, что такая модель представляет собой контракт между программным обеспечением и памятью. Почему необходим такой контракт?
10. Рассмотрим мультипроцессор с общей шиной. Что произойдет, если два процессора попытаются получить доступ к глобальной памяти в один и тот же момент?
11. Рассмотрим мультипроцессор с общей шиной. Что произойдет, если три процессора попытаются получить доступ к глобальной памяти в один и тот же момент?

12. Предположим, что по техническим причинам следящий кэш может следить только за адресными линиями, а за информационными — нет. Повлияет ли это изменение на протокол сквозной записи?
13. Рассмотрим простую модель мультипроцессорной системы с шиной и без кэширования. Предположим, что одна из каждых четырех команд обращается к памяти, причем при каждом обращении к памяти шина занимается на все время выполнения команды. Если шина занята, то запрашивающий процессор ставится в очередь FIFO. Насколько быстрее будет работать система с 64 процессорами по сравнению с однопроцессорной системой?
14. Протокол MESI имеет четыре состояния. Другой протокол согласования кэшей при отложенной записи имеет три состояния. Каким из состояний протокола MESI можно пожертвовать и каковы будут последствия каждого из четырех вариантов? Если бы вам пришлось выбрать только три состояния, какие бы вы выбрали?
15. Бывают ли в протоколе MESI такие ситуации, когда строка кэша присутствует в локальной кэш-памяти, но при этом все равно требуется транзакция шины? Если да, то опишите такую ситуацию.
16. Предположим, что к общей шине подсоединено n процессоров. Вероятность того, что один из процессоров пытается использовать шину в данном цикле, равна p. Какова вероятность, что:
    1) шина свободна (0 запросов);
    2) совершается один запрос;
    3) совершается более одного запроса.
17. Назовите главное преимущество и главный недостаток перекрестной коммутации.
18. Сколько схем перекрестной коммутации в полноценном процессоре Fire E25K компании Sun?
19. Предположим, что провод между коммутатором 2A и коммутатором 3B в сети omega поврежден. Какие именно элементы будут отрезаны друг от друга?
20. «Горячие» точки (области памяти, к которым часто происходят обращения) в сетях с многоступенчатой коммутацией представляют собой серьезную проблему. А являются ли они проблемой в системах с шинной организацией?
21. Сеть omega соединяет 4096 RISC-процессоров, время цикла каждого из которых составляет 60 нс, с 4096 бесконечно быстрыми модулями памяти. Каждый коммутирующий элемент дает задержку 5 нс. Сколько слотов отсрочки требуется для команды `LOAD`?
22. Рассмотрим машину, использующую сеть omega (см. рис. 8.27). Предположим, что программа и стек i хранятся в модуле памяти i. Какое незначительное изменение топологии может значительно повлиять на производительность? (Эта модифицированная топология используется в IBM RP3 и BBN Butterfly.) Какой недостаток имеет новая топология по сравнению со старой?
23. В NUMA-мультипроцессоре обращение к локальной памяти занимает 20 нс, а к памяти другого процессора — 120 нс. Программа во время выполнения совершает N обращений к памяти, 1 % из которых — обращения к странице P.

Изначально эта страница находится в удаленной памяти, а на копирование ее из локальной памяти требуется C нс. При каких обстоятельствах эту страницу следует копировать локально, если ее не используют другие процессоры?

24. Рассмотрим CC-NUMA-мультипроцессор, такой как на рис. 8.29, но содержащий 512 узлов по 8 Мбайт каждый. Если длина строки кэша составляет 64 байта, каков процент непроизводительных затрат для каталогов? Как повлияет увеличение числа узлов на непроизводительные затраты (они увеличатся, уменьшатся или останутся без изменений)?
25. Чем модель NC-NUMA отличается от CC-NUMA?
26. Вычислите диаметр сети для каждой из топологий, представленных на рис. 8.33.
27. Для каждой из топологий, представленных на рис. 8.33, определите коэффициент отказоустойчивости (максимальное число линий связи, после утраты которых сеть не окажется разделена на две части).
28. Рассмотрим топологию двойной тор (см. рис. 8.33, *e*), расширенную до размера $k \times k$. Каков диаметр такой сети? (*Подсказка*: четное и нечетное значение k нужно рассматривать отдельно.)
29. Представим сеть в форме куба $8 \times 8 \times 8$. Каждая линия связи имеет дуплексную пропускную способность 1 Гбайт/с. Какова пропускная способность сечения в этой сети?
30. Закон Амдала ограничивает потенциальное ускорение, достижимое в параллельном компьютере. Вычислите как функцию от f максимально возможное ускорение, если число процессоров стремится к бесконечности. Каково значение этого предела для $f = 0{,}1$?
31. Рисунок 8.44 призван показать, что масштабирование в случае с шиной невозможно, а в случае с решеткой возможно и желательно. Предположим, каждая шина или линия связи имеет пропускную способность b. Вычислите среднюю пропускную способность на каждый процессор для каждого из четырех случаев. Затем масштабируйте каждую систему до 64 процессоров и выполните те же вычисления. Чему равен предел, если число процессоров стремится к бесконечности?
32. В этой книге мы обсуждали три варианта примитива `send` — синхронный, блокирующий и неблокирующий. Предложите четвертый вариант, напоминающий блокирующий, но немного отличающийся по свойствам. Какое преимущество и каков недостаток имеет новый примитив по сравнению с обычной блокирующей операцией `send`?
33. Рассмотрим компьютер, который работает в сети с аппаратным широковещанием (например, Ethernet). Почему важно соотношение операций чтения (которые не изменяют внутреннее состояние переменных) и записи (которые изменяют внутреннее состояние переменных)?

Глава 9
Библиография

В этой главе приведен алфавитный список всех книг и статей, на которые в этой книге есть ссылки.

1. *Abramson, D.*: «Mixing Cloud and Grid Resources for ManyTask Computing», Proc. Int'l Workshop on Many Task Computing on Grids and Supercomputers,ACM, pp. 1–2, 2011.
2. *Adams, M., and Dulchinos, D.* «OpenCable», IEEE Commun. Magazine, vol. 39, pp. 98–105, June 2001.
3. *Adiga, N.R. et al.* «An Overview of the BlueGene/L Supercomputer», Proc. Supercomputing 2002, ACM, pp. 1–22, 2002.
4. *Adve, S. V., and Hill, M.* «Weak Ordering: A New Definition», Proc. 17th Ann. Int'l. Symp. on Computer Arch., ACM, pp. 2–14, 1990.
5. *Agerwala, T., and Cocke, J.* «High Performance Reduced Instruction Set Processors», IBM T.J. Watson Research Center Technical Report RC12434, 1987.
6. *Ahmadinia, A., and Shahrabi, A.*: «A Highly Adaptive and Efficient Router Architecture for Network-on-Chip», Computer J.,vol. 54, pp. 1295–1307, Aug. 2011.
7. *Alam, S., Barrett, R., Bast, M., Fahey, M.R., Kuehn, J., McCurdy, Rogers, J., Roth, P., Sankaran, R., Vetter, J.S., Worley,P., and Yu,W.*: «Early Evaluation of IBM BlueGene/P», Proc. ACM/IEEE Conf.onSupercomputing,ACM/IEEE, 2008.
8. *Alameldeen, A.R., and Wood, D.A.* «Adaptive Cache Compression for High-Performance Processors». Proc. 31st Ann. Int'l Sym. on Computer Arch. ACM, pp. 212–223, 2004.
9. *Almasi, G. S. et al.* «System Management in the BlueGene/L Supercomputer», Proc. 17th Int'l Parallel and Distr. Proc. Symp., IEEE, 2003a.
10. *Almasi, G.S. et al.* «An Overview Of The Bluegene/L System Software Organization», Par. Proc. Letters, vol. 13, 561–574, April 2003b.
11. *Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.* «TreadMarks: Shared Memory Computing on a Network of Workstations», IEEE Computer Magaxine, vol. 29, pp. 18–28, Feb. 1996.
12. *Anderson, D.* Universal Serial Bus System Architecture, Reading, MA: Addison-Wesley, 1997.
13. *Anderson, D., Budruk, R., and Shanley, T.* PCI Express System Architecture, Reading, MA: Addison-Wesley, 2004.
14. *Anderson, T. E., Culler, D. E., Patterson, D. A., and the NOW team* «A Case for NOW (Networks of Workstations)», IEEE Micro Magazine, vol. 15, pp. 54–64, Feb. 1995.

15. *August, D. I., Connors, D. A., Mahlke, S. A., SIAS, J. W., Crozier, K. M., Cheng, B. -C., Eaton, P. R., Olaniran, Q. B., and HWU, W. -M.* «Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM, pp. 227–237, 1998.
16. *Bal, H. E.* Programming Distributed Sysytems, Hemel Hempstead, England: Prentice Hall Int'l., 1991.
17. *Bal, H. E., Bhoedjang, R., Hofman, R, Jacobs, C., Langendoen, K., Ruhl, T., and Kaashoek, M. F.* «Performance Evaluation of the Orca Shared Object System», ACM Trans. on Computer Systems, vol. 16, pp. 1–40, Feb. 1998.
18. *Bal, H. E., Kaashoek, M.F., and Tanenbaum, A. S.* «Orca: A Language for Parallel Programming of Distributed Systems», IEEE trans. on Software Engeneering, vol. 18, pp. 190–205, March 1992.
19. *Bal, H. E., and Tanenbaum, A. S.* «Distributed Programming with Shared Data», Proc. 1988 Int'l. Conf. on Computer Languages, IEEE, pp. 82–91, 1988.
20. *Balasangameshwara, J., and Raju, N.*: «A Hybrid Policy for Fault Tolerant Load Balancing in Grid Computing Environments», J. Network and Computer Applications, vol. 35, pp. 412–422, Jan. 2012.
21. *Barroso, L.A., Dean, J., Holzle, U.* «Web Search for a Planet: The Google Cluster Architecture», IEEE Micro Magazine, vol. 23, pp. 22–28, March-April 2003.
22. *Bechini, A., Conte, T.M., and Prete, C.A.* «Opportunities and Challenges in Embedded Systems», IEEE Micro Magazine, vol. 24, pp. 8–9, July-Aug. 2004.
23. *Bhakthavatchalu, R., Deepthy, G.R.; and Shanooja, S.*: «Implementation of Reconfigurable Open Core Protocol Compliant Memory System Using VHDL», Proc. Int'l Conf. on Industrial and Information Systems, pp. 213–218, 2010.
24. *Bjornson, R. D.* «Linda on Distributed Memory Multiprocessors», Ph. D. Thesis, Yale Univ., 1993.
25. *Blumrich, M., Chen, D., Chiu, G., Coteus, P., Gara, A., Giampapa, M.E., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopcsay, G.V., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vransas, P., and Liebsch, T.* «An Overview of the BlueGene/L System», IBM J. Research and Devel., vol. 49, March–May, 2005.
26. *Bose, P.* «Computer architecture research: Shifting priorities and newer challenges», IEEE Micro Magazine, vol. 24, p. 5, Nov–Dec. 2004.
27. *Bouknight, W. J., Denenberg, S. A., Mcintyre, D. E., Randall, J. M., Sameh, A. H., and Slotnick, D. L.* «The Illiac IV System», Proc. IEEE, pp. 369–388, April 1972.
28. *Bradley,D.*: «A Personal History of the IBM PC», IEEE Computer, vol. 44, pp. 19–25, Aug. 2011.
29. *Bride, E.*: «The IBM Personal Computer: A Software-DrivenMarket», IEEE Computer, vol. 44, pp. 34–39, Aug. 2011.
30. *Brightwell, R., Camp, W., Cole, B., DeBenedictis, Leland, R, and Tompkins, J.* «Architectural Specification for Massively Parallel Computers — An Experience and Measurement-Based Approach», Concurrency and Computation: Practice and Experience,vol. 17, pp. 1–46, 2005.

31. *Brightwell, R., Underwood, K.D., Vaughan, C., and Stevenson, J.*: «Performance Evaluation of the Red Storm Dual-Core Upgrade,'' Concurrency and Computation: Practice and Experience,vol. 22, pp. 175–190, Feb. 2010.
32. *Bryant, R.E., and O'Hallaron, D.* Computer Systems: A Programmer's Perspective Upper Saddle River, NJ: Prentice Hall, 2003.
33. *Burkhardt, H., Frank, S., Knobe, B., and Rothnie, J.* «Overview of the KSR-1 Computer Sysytem», Technical Report KSR-TR-9202001, Kendall Square Research Corp, Cambridge, MA, 1992.
34. *Carriero, N., and Gelernter, D.* «Linda and Context», Commun. of the ACM, vol. 32, pp. 444–458, April 1989.
35. *Celaya, J., and Arronategui, U.*: «A Highly Scalable Decentralized Scheduler of Tasks with Deadlines», Proc. 12th Int'l Conf. on Grid Computing, IEEE/ACM, pp. 58–65, 2011.
36. *Charlesworth, A.* «The Sun Fireplane Interconnect», IEEE Micro Magazine, vol. 22, pp. 36–45, Jan.-Feb. 2002.
37. *Charlesworth, A.* «The Sun Fireplane Interconnect», Proc. Conf. on High Perf. Networking and Computing, ACM, 2001.
38. *Charlesworth, A., Phelps, A., Williams, R., and Gilbert, G.* «Gigaplane-XB: Extending the Ultra Enterprise Family», Proc. Hot Interconnects V, IEEE, 1998.
39. *Chen, L., Dropsho, S., Albonesi, D.H.* «Dynamic Data Dependence Tracking and its Application to Branch Prediction», Proc. Ninth Int'l Symp. on High-Performance Computer Arch., IEEE, pp. 65–78, 2003.
40. *Cheng, L., and Carter, J.B.*: «Extending CC-NUMA Systems to Support Write Update Optimizations», Proc. 2008 ACM/IEEE Conf. on Supercomputing, ACM/IEEE, 2008.
41. *Chou, Y., Fahs, B., Abraham, S.* «Microarchitecture Optimizations for Exploiting Memory-Level Parallelism, Proc. 31st Ann. Int'l Symp. on Computer Arch., ACM, pp. 76–77, 2004.
42. *Cohen, D.* «On Holy Wars and a Plea for Peace», IEEE Computer Magazine, vol. 14, pp. 48–54, Oct. 1981.
43. *Corbaty, F.J., and Vyssotsky, V. A.* «Introduction and Overview of the MULTICS System», Proc. FJCC, pp. 185–196, 1965.
44. *Denning, P. J.* «The Working Set Model for Program Behavior», Commun. of the ACM, vol. 11, pp. 323–333, May 1968.
45. *Dijkstra, E. W.* «GOTO Statement Considered Harmful», Commun. of the ACM, vol. 11, pp. 147–148, Mar. 1968a.
46. *Dijkstra, E. W.* «Co-operating Sequential Processes», in Programming Languages, F. Genuys (ed.), New York: Academic Press, 1968b.
47. *Donaldson, G., and Jones, D.* «Cable Television Broadband Network Architectures», IEEE Commun. Magazine, vol. 39, pp. 122–126, June 2001.
48. *Dubois, M., Scheurich, C., and Briggs, F .A.* «Memory Access Buffering in Multiprocessors», Proc. 13th Ann. Int'l. Symp. on Computer Arch., ACM, pp. 434–442, 1986.

49. *Dulong, C.* «The IA-64 Architecture at Work», IEEE Computer Magazine, vol. 31, pp. 24–32, July 1998.
50. *Dutta-Roy, A.* «An Overview of Cable Modem Technology and Market Perspectives», IEEE Commun. Magazine, vol. 39, pp. 81–88, June 2001.
51. *Faggin, F., Hoff, M. E., Jr., Mazor, S., and Shima, M.* «The History of the 4004», IEEE Micro Magazine, vol. 16, pp. 10–20, Nov. 1996.
52. *Falcon, A., Stark, J., Ramirez, A., Lai, K., and Valero, M.* «Prophet/Critic Hybrid Branch Prediction», Proc. 31th Ann. Int'l Symp. on Computer Arch., ACM, pp. 250–261, 2004.
53. *Fisher, J. A., and Freudenberger, S. M.* «Predicting Conditional Branch Directions from Previous Runs of a Program», Proc. 5th Conf. on Arch. Support for Prog. Lang. and Operating Syst., ACM, pp. 85–95, 1992.
54. *Flynn, D.* «AMBA: Enabling Reusable On-Chip Designs», IEEE Micro Magazine, vol. 17, pp. 20–27, July 1997.
55. *Flynn, M. J.* «Some Computer Organizations and Their Effectiveness», IEEE Trans. on Computers, vol. C-21, pp. 948–960, Sept. 1972.
56. *Foster, I., and Kesselman, C.* The Grid 2: Blueprint for a New Computing Infrastructure, San Francisco: Morgan Kaufman, 2003.
57. *Fotheringham, J.* «Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store», Commun. of the ACM, vol. 4, pp. 435–436, Oct. 1961.
58. *Freitas, H.C., Madruga, F.L., Alves, M., and Navaux, P.*: «Design of Interleaved Multithreading for Network Processors on Chip», Proc. Int'l Symp. on Circuits and Systems,IEEE, 2009.
59. *Gaspar, L., Fischer, V., Bernard, F., Bossuet,L., and Cotret, P.*: «Hcrypt: A Novel concept of Crypto-Processor with Secured Key Management," Int'l Conf.On Reconfigurable Computing And FPGAs, 2010.
60. *Gaur, J., Chaudhuri, C., And Subramoney, S.*: «Bypass and Insertion Algorithms for Exclusive Last-Level caches», Proc. 38th Int'l Symp. On Computer Arch.,ACM, 2011.
61. *Gebhart, M., Johnson, D.R., Tarjan, D., Keckler, S.W., Dally, W.J.,Lindholm, E., And Skadron, K.*: «Energy-Efficient Mechanisms For Managing Thread Context In Throughput Processors», Proc. 38th Int'l Symp. On Computer Arch. ACM, 2011.
62. *Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderram, V.* PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing, Cambridge, MA: M.I.T. Press, 1994.
63. *Gepner, P., Gamayunov, V., and Fraser, D.L.*: «The 2nd Generation Intel Core Processor. Architectural Features Supporting HPC», Proc. 10th Int'l Symp. on Parallel and Dist. Computing, pp. 17–24, 2011.
64. *Gerber, R., and Binstock, A.* Programming with Hyper-Threading Technology, Santa Clara, CA: Intel Press, 2004.
65. *Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P.B., Gupta, A., and Hennessy, J.L.*: «Memory Consistency and Event Ordering in Scalable Shared-Memory

Multiprocessors», Proc. 17th Ann. Int'l Symp. on Comp. Arch.,ACM, pp. 15–26, 1990.

66. *Ghemawat, S., Gobioff, H., and Leung, S.-T.* «The Google File System,» Proc. 19th Symp. on Operating Systems Principles, ACM, pp. 29–43, 2003.
67. *Goodman, J. R.* «Using Cache Memory to Reduce Processor Memory Traffic», Proc. 10th Ann. Int'l. Symp. on Computer Arch., ACM, pp. 124–131, 1983.
68. *Goodman J. R.* «Cache Consistency and Sequential Consistency», Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.
69. *Goth, G.*: «IBM PC Retrospective:There Was Enough Right to Make ItWork», IEEE Computer,vol. 44, pp. 26–33, Aug. 2011.
70. *Gropp, W., Lusk, E., and Skjellum, A.* «Using MPI: Portable Parallel Programming with the Message Passing Interface», Cambridge, MA: M.I.T. Press, 1994.
71. *Gupta, N., Mandal, S., Malave, J., Mandal, A., and Mahapatra, R.N.*: «A Hardware Scheduler for Real Time Multiprocessor System on Chip», Proc. 23rd Int'l Conf. on VLSI Design, IEEE, 2010.
72. *Gurumurthi, S., Sivasubramaniam, Kandemir, M., and Franke, H.* «Reducing Disk Power Consumption in Servers with DRPM», IEEE Computer Magazine, vol. 36, pp. 59–66, Dec. 2003.
73. *Hagersten, E., Landin, A., Haridi, S.* «DDM — A Cache-Only Memory Architecture», IEEE Computer Magazine, vol. 25, pp. 44–54, Sept. 1992.
74. *Haghighizadeh, F., Attarzadeh, H., and Sharifkhani, M.*: «A Compact 8-Bit AES Crypto-processor», Proc. Second. Int'l Conf. on Computer and Network Tech., IEEE, 2010.
75. *Hamming, R. W.* «Error Detecting and Error Correcting Codes», Bell Syst. Tech. J., vol. 29, pp. 147–160, April 1950.
76. *Henkel, J., Hu, X.S., and Bhattacharyya, S.S.* «Taking on the Embedded System Challenge», IEEE Computer Magazine, vol. 36, pp. 35–37, April 2003.
77. *Hennessy, J. L.* «VLSI Processor Architecture», IEEE Trans. on Computers, vol. C-33, pp. 1221–1246, Dec. 1984.
78. *Herrero, E., Gonzalez, J., and Canal, R.*: «Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchyfor Chip Multiprocessors», Proc. 23rd Int'l Conf. on VLSI Design, IEEE, 2010.
79. *Hoare, C. A. R.* «Monitors, An Operating System Structuring Concept», Commun. of the ACM, vol. 17, pp. 549–557, Oct. 1974; Erratum in Commun. of the ACM, vol. 18, p. 95, Feb.1975.
80. *Hwu, W. -M.* «Introduction to Predicated Execution», IEEE Computer Magazine, vol. 31, pp. 49–50, Jan. 1998.
81. *Jimenez, D.A.* «Fast Path-Based Neural Branch Prediction», Proc. 36th Int'l Symp. on Microarchitecture, IEEE., pp. 243–252, 2003.
82. *Johnson, K. L., Kaashoek, M. F., and Wallach, D. A.* «CRL: High-Performance All-Software Distributed Shared Memory», Proc. 15th Symp. on Operating Systems Principles, ACM, pp. 213–228, 1995.
83. *Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Ahn, J.H., Mattson, P., and Owens, J.D.* «Programmable Stream Processors», IEEE Computer Magazine, vol. 36, pp. 54–62, Aug. 2003.

84. *Kaufman, C., Perlman, R., and Speciner, M.* «Network Security», 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2002.
85. *Kim, N.S., Austin, T., Blaauw, D., Mudge, T., Flautner, K., Hu, J.S., Irwin, M.J., Kandemir, M., and Narayanan, V.* «Leakage Current: Moore's Law Meets Static Power», IEEE Computer Magazine, vol. 36, pp. 68–75, Dec. 2003.
86. *Knuth, D. E.*: «The Art of Computer Programming: Fundamental Algorithms», 3rd ed.. Reading, MA: Addison-Wesley, 1997.
87. *Kontothanassis, L., Hunt, G., Stets, R., Hardavellas, N., Cierniad, M., Parthasarathy, S., Meira, W., Dwarkadas, S., and Scott, M.*: «VM-Based Shared Memory on Low Latency Remote Memory Access Networks», Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, pp. 157–169, 1997.
88. *Lamport, L.* «How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs», IEEE Trans. on Computers, vol. C-28, pp. 690–691, Sept. 1979.
89. *LaRowe, R. P., and Ellis, C. S.* «Experimental Comparison of Memory Management Policies for NUMA Multiprocessors», ACM Trans. on Computer Systems, vol. 9, pp. 319–363, Nov. 1991.
90. *Lee, J., Keleher, P., and Sussman, A.*: «Supporting Computing Element Heterogeneity in P2P Grids», Proc. IEEE Int'l Conf.on Cluster Computing, IEEE, pp. 150–158, 2011.
91. *Li, K., and Hudak, P.* «Memory Coherence in Shared Virtual Memory Systems», ACM Trans. on Computer Systems, vol. 7, pp. 321–359, Nov. 1989.
92. *Lin, Y.-N., Lin, Y.-D., and Lai, Y.-C.*: «Thread Allocation in CMP-based Multithreaded Network Processors», Parallel Computing,vol. 36, pp. 104–116, Feb. 2010.
93. *Lu, H., Cox, A. L., Dwarkadas, S., Rajamony, R., and Zwaenepoel, W.* «Software Distributed Shared Memory Support for Irregular Applications», Proc. 6th Conf. on Prin. and Practice of Parallel Progr., pp. 48–56, June 1997.
94. *Lukasiewicz, J.*: «Aristotle's Syllogistic», 2nd ed., Oxford: Oxford University Press, 1958.
95. *Lyytinen, K., and Yoo, Y.*: «Issues and Challenges in Ubiquitous Computing», Commun. of the ACM, vol. 45, pp. 63–65, Dec. 2002.
96. *Martin, R. P., Vahdat, A. M., Culler, D. E., and Anderson, T. E.* «Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture», Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, pp. 85–97, 1997.
97. *Mayhew, D., and Krishnan, V.* «PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects», Proc. 11th Symp. on High Perf. Interconnects IEEE, pp. 21–29, Aug. 2003.
98. *McKusick, M. K., Bostic, K., Karels, M., and Quarterman, J. S.* «The Design and Implementation of the 4.4 BSD Operating System», Reading, MA: Addison-Wesley, 1996.
99. *McNairy, C., and Soltis, D.* «Itanium 2 Processor Microarchitecture», IEEE Micro Magazine, vol. 23, pp. 44–55, March–April 2003.
100. *Mishra, A.K., Vijaykrishnan, N., and Das, C.R.*: «A Case for Heterogeneous On-Chip Interconnects for CMPs», Proc. 38th Int'l Symp. on Computer Arch. ACM, 2011.

101. *Morgan, C.*: «Portraits in Computing», New York: ACM Press, 1997.
102. *Moudgill, M., and Vassiliadis, S.* «Precise Interrupts», IEEE Micro Magazine, vol. 16, pp. 58–67, Feb. 1996.
103. *Mullender, S. J., and Tanenbaum, A. S.* «Immediate Files», Software — Practice and Experience, vol. 14, pp. 365–368, 1984.
104. *Naeem, A., Chen, X., Lu, Z., and Jantsch, A.*: «Realization and Performance Comparison of Sequential and Weak Memory Consistency Models in Network-On-Chip Based Multicore Systems», Proc. 16th Design Automation Conf.Asia and South Pacific, IEEE, pp. 154–159, 2011.
105. *Organick, E.*: «The MULTICS System», Cambridge, MA: M.I.T. Press, 1972.
106. *Oskin, M., Chong, F.T., and Chuang, I.L.*: «A Practical Architecture for Reliable Quantum Computers», IEEE Computer Magazine, vol. 35, pp. 79–87, Jan. 2002.
107. *Papamarcos, M., and Patel., J.* «A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories», Proc. 11th Ann. Int'l. Symp. on Computer Arch., ACM, pp. 348–354, 1984.
108. *Parikh, D., Skadron, K., Zhang, Y., and Stan, M.* «Power-Aware Branch Prediction: Characterization and Design», IEEE Trans. on Computers, vol. 53, pp. 168–186, Feb. 2004.
109. *Patterson, D. A.* «Reduced Instruction Set Computers», Commun. of the ACM, vol. 28, pp. 8–21, Jan. 1985.
110. *Patterson, D. A., Gibson, G., and Katz, R.* «A case for redundant arrays of inexpensive disks (RAID)», Proc. ACM SIGMOD Int'l. Conf. on Management of Data, ACM, pp. 109–166, 1988.
111. *Patterson, D. A., and Sequin, C. H.* «A VLSI RISC», IEEE Computer Magazine, vol. 15, pp. 8–22, Sept. 1982.
112. *Pountain, D.* «Pentium: More RISC than CISC», Byte, vol. 18, pp. 195–204, Sept. 1993.
113. *Radin, G.* «The 801 Minicomputer», Computer Arch. News, vol. 10, pp. 39–47, March 1982.
114. *Raman, S.K., Pentkovski, V., and Keshava, J.* «Implementing Streaming SIMD Extensions on the Pentium III Processor», IEEE Micro Magazine, vol. 20, pp. 47–57, July–Aug. 2000.
115. *Ritchie, D.M.*: «Reflections on Software Research», Commun. of the ACM, vol. 27, pp. 758–760, Aug. 1984.
116. *Ritchie, D. M., and Thompson, K.* «The UNIX Time-Sharing System», Commun. of the ACM, vol. 17, pp. 365–375, July 1974.
117. *Robinson, G.S.* «Toward the Age of Smarter Storage», IEEE Computer Magazine, vol. 35, pp. 35–41, Dec. 2002.
118. *Rosenblum, M., and Ousterhout, J. K.* «The Design and Implementation of a Log-Structured File System», Proc. Thirteenth Symp. on Operating System Principles, ACM, pp. 1–15, 1991.
119. *Russinovich, M.E., and Solomon, D.A.* Microsoft Windows Internals, 4th ed., Redmond, WA: Microsoft Press, 2005.

120. *Rusu, S., Muljono, H., and Cherkauer, B.* «Itanium 2 Processor 6M», IEEE Micro Magazine, vol. 24, pp. 10–18, March–April 2004.
121. *Saha, D., and Mukherjee, A.* «Pervasive Computing: A Paradigm for the 21st Century», IEEE Computer Magazine, vol. 36, pp. 25–31, March 2003.
122. *Sakamura, K.* «Making Computers Invisible», IEEE Micro Magazine, vol. 22, pp. 7–11, 2002.
123. *Sanchez, D., and Kozyrakis, C.*: «Vantage: Scalable and Efficient Fine-Grain Cache Partitioning», Proc. 38th Ann. Int'l Symp. on Computer Arch.,ACM, pp. 57–68, 2011.
124. *Scales, D. J., Gharachorloo, K., and Thekkath, CA.* «Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory», Proc. 7th Int'l. Conf. on Arch. Support for Prog. Long. and Oper. Syst., ACM, pp. 174–185, 1996.
125. *Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C.* «An Implementation of a Log-Structured File System for UNIX», Proc. Winter 1993 USENIX Technical Conf., pp. 307–326, 1993.
126. *Shanley, T., and Anderson, D.* PCI System Architecture, 4th ed.. Reading, MA: Addison–Wesley, 1999.
127. *Shoufan, A., Huber, N., and Molter, H.G.*: «ANovelCryptoprocessor Architecture for Chained Merkle Signature Schemes», Microprocessorsand Microsystems,vol. 35, pp. 34–47, Feb.2011.
128. *Singh, G.*: «The IBM PC: The Silicon Story», IEEE Computer,vol. 44, pp. 40–45, Aug. 2011.
129. *Slater, R.* Portraits in Silicon, Cambridge, MA: M.I.T. Press, 1987.
130. *Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., and Dongarra, J.*: «MPI: The Complete Reference Manual», Cambridge, MA: M.I.T. Press, 1996.
131. *Solari, E., and Congdon, B.*: «PCI Express Design & System Architecture», Research Tech, Inc., 2005.
132. *Solari, E., and Willse, G.*: «PCI and PCI-X Hardware and Software», 6th ed., San Diego, CA: Annabooks, 2004.
133. *Sorin, D.J., Hill, M.D., and Wood, D.A.*: «A Primer on Memory Consistency and Cache Coherence», San Francisco: Morgan&Claypool, 2011.
134. *Stets, R., Dwarkadas, S., Hardavellas, N., Hunt, G., Kontothanassis, L., Parthasarathy, S., and Scott, M.* «CASHMERE-2L: Software Coherent Shared Memory on Clustered Remote-Write Networks», Proc. 16th Symp. on Operating Systems Principles, ACM, pp. 170–183, 1997.
135. *Summers, C.K.*: «ADSL: Standards, Implementation, and Architecture», Boca Raton, FL: CRC Press, 1999.
136. *Sunderram, V. B.*: «PVM: A Framework for Parallel Distributed Computing», Concurrency: Practice and Experience, vol. 2, pp. 315–339, Dec. 1990.
137. *Swan, R. J., Fuller, S. H., and Siewiorek, D. P.* «Cm* —A Modular Multiprocessor», Proc. NCC, pp. 645–655, 1977.
138. *Tan, W. M.*: «Developing USB PC Peripherals», San Diego, CA: Annabooks, 1997.

139. *Tanenbaum, A. S., and Wetherall, D. J.*: «Computer Networks», 5th ed., Upper Saddle River, NJ: Prentice Hall, 2011.
140. *Thompson, K.*: «Reflections on Trusting Trust», Commun. of the ACM,vol. 27, pp. 761–763, Aug. 1984.
141. *Thompson, J., Dreisigmeyer, D.W., Jones, T., Kirby, M., and Ladd, J.*: «Accurate Fault Prediction of BlueGene/P RAS Logs via Geometric Reduction», IEEE, pp. 8–14, 2010.
142. *Treleaven, P.* «Control-Driven, Data-Driven, and Demand-Driven Computer Architecture», Parallel Computing, vol. 2, 1985.
143. *Tu, X., Fan, X., Jin, H., Zheng, L., and Peng, X.*: «Transactional Memory Consistency: A New Consistency Model for Distributed Transactional Memory», Proc. Third Int'l Joint Conf. on Computational Science and Optimization,IEEE, 2010.
144. *Vahalia, U.*: «UNIX Internals», Upper Saddle River, NJ: Prentice Hall, 1996.
145. *Vahid, F.*: «The Softening of Hardware», IEEE Computer Magazine, vol. 36, pp. 27-34, April 2003.
146. *Vetter, P., Goderis, D., Verpooten, L., and Granger, A.* «Systems Aspects of APON/VDSL Deployment», IEEE Commun. Magazine, vol. 38, pp. 66–72, May 2000.
147. *Vu, T. D., Zhang, L., and Jesshope, C.*: «The Verification of the On-Chip COMA Cache Coherence Protocol», Proc. 12th Int'l Conf.onAlgebraic Methodology and Software Technology,Springer-Verlag, pp. 413–429, 2008.
148. *Weiser, M.* «The Computer for the 21st Century», IEEE Pervasive Computing, vol. 1, pp. 19–25, Jan.–March 2002; originally published in Scientific American, Sept. 1991.
149. *Wilkes, M. V.* «Computers Then and Now», J. ACM, vol. 15, pp. 1–7, Jan. 1968.
150. *Wilkes, M. V.* «The Best Way to Design an Automatic Calculating Machine», Proc. Manchester Univ. Computer Inaugural Conf., 1951.
151. *Wing-Kei, Y., Huang, R., Xu, S., Wang, S.-E., Kan, E., and Suh, G.E.*: «SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained MultiThreading Architectures», Proc. 38th Int'l Symp. on Computer Arch. ACM, 2011.
152. *Yamamoto, S., and Nakao, A.*: «Fast Path Performance of Packet Cache Router Using Multi-core Network Processor», Proc. Seventh Symp. on Arch.for Network and Comm. Sys., ACM/IEEE, 2011.
153. *Zhang, L., and Jesshope, C.*: «On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores», Proc. of 2007 European Conf.onParallel Processing, Springer-Verlag, pp. 38–48, 2008.

Приложение А
Двоичные числа

Арифметика, применяемая в компьютерах, отличается от арифметики, к которой мы все привыкли. Во-первых, компьютеры оперируют числами, точность которых конечна и фиксирована. Во-вторых, в большинстве компьютеров используется не десятичная, а двоичная система счисления. Эти две темы и рассматриваются в этом приложении.

Числа конечной точности

Когда люди выполняют какие-либо арифметические действия, их не волнует вопрос, сколько десятичных разрядов занимает то или иное число. Физики, к примеру, могут вычислить, что во вселенной существуют 1078 электронов, и их не волнует тот факт, что полная запись этого числа потребует 79 десятичных разрядов. Проблема нехватки бумаги для записи числа никогда не возникает.

С компьютерами дело обстоит иначе. В большинстве компьютеров количество доступной памяти для хранения чисел фиксировано и зависит от того, когда был выпущен этот компьютер. Если приложить усилия, программист сможет представлять числа в два, три и более раз большие, чем позволяет объем памяти, но это не меняет природы данной проблемы. Память компьютера ограничена, поэтому мы можем иметь дело только с такими числами, которые можно представить в фиксированном количестве разрядов. Такие числа называются **числами конечной точности**.

Рассмотрим ряд положительных целых чисел, которые можно записать тремя десятичными разрядами без десятичной точки и без знака. В этот ряд входит ровно 1000 чисел: 000, 001, 002, 003, ..., 999. При таком ограничении невозможно выразить определенные типы чисел. Сюда входят:

- числа больше 999;
- отрицательные числа;
- дроби;
- иррациональные числа;
- комплексные числа.

Одно из свойств набора всех целых чисел — **замкнутость** по отношению к операциям сложения, вычитания и умножения. Другими словами, для каждой пары целых чисел i и j числа $i + j$, i $- j$ и $i \times j$ — тоже целые числа. Ряд целых чисел не замкнут относительно деления, поскольку существуют такие значения i и j, для которых i/j не выражается в виде целого числа (например, 7/2 или 1/0).

Числа конечной точности не замкнуты относительно всех четырех операций. Вот примеры операций над трехразрядными десятичными числами.

- Слишком большое число:

$$600 + 600 = 1200.$$

- Отрицательное число:

$$003 - 005 = -2.$$

- Слишком большое число:

$$050 \times 050 = 2500.$$

- Не целое число:

$$007 / 002 = 3{,}5.$$

Отклонения можно разделить на два класса: операции, результат которых больше самого большого числа ряда (ошибка переполнения) или меньше самого маленького числа ряда (ошибка потери значимости), и операции, результат которых не является слишком маленьким или слишком большим, а просто не является членом ряда. Из четырех приведенных примеров первые три относятся к первому классу, а четвертый — ко второму классу.

Поскольку объем памяти компьютера ограничен, и компьютер должен выполнять арифметические действия над числами конечной точности, с точки зрения классической математики результаты определенных вычислений оказываются неправильными. Ошибка в данном случае — это только следствие конечной природы представления чисел в вычислительном устройстве. Некоторые компьютеры имеют встроенную аппаратную поддержку для обнаружения ошибок переполнения.

Алгебра чисел конечной точности отличается от обычной алгебры. В качестве примера рассмотрим ассоциативный закон

$$a + (b - c) = (a + b) - c.$$

Вычислим обе части выражения для $a = 700$, $b = 400$ и $c = 300$. В левой части сначала вычислим значение $(b - c)$. Оно равно 100. Затем прибавим это число к a и получим 800. Чтобы вычислить правую часть, сначала вычислим $(a + b)$. Для 3-разрядных целых чисел получится переполнение. Результат будет зависеть от компьютера, но он окажется неравным 1100. Вычитание 300 из какого-то числа, отличного от 1100, не даст в результате 800. Ассоциативный закон не имеет силы. Важна очередность выполнения операций.

Другой пример — дистрибутивный закон:

$$a \times (b - c) = a \times b - a \times c.$$

Вычислим обе части выражения для $a = 5$, $b = 210$ и $c = 195$. В левой части $5 \times 15 = 75$. В правой части 75 не получается, поскольку результат выполнения операции $a \times b$ выходит за пределы ряда.

Исходя из этих примеров, кто-то может сделать вывод, что компьютеры совершенно непригодны для выполнения арифметических действий. Вывод, естественно, неверен, но эти примеры наглядно показывают, как важно понимать механизм работы компьютера и знать о его ограничениях.

Позиционные системы счисления

Обычное десятичное число состоит из цепочки десятичных разрядов и иногда десятичной точки (запятой). Общая форма записи показана на рис. А.1. Десятка выбрана в качестве основы возведения в степень (и называется **основанием системы счисления**), поскольку мы используем 10 цифр. В компьютерах удобнее иметь дело с другими основаниями системы счисления. Самые важные из них — 2, 8 и 16. Соответствующие системы счисления называются **двоичной**, **восьмеричной** и **шестнадцатеричной**.

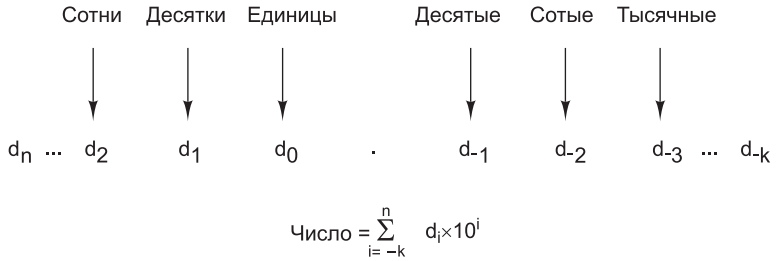


Рис. А.1. Общая форма десятичного числа

Система счисления с основанием k требует k различных символов для записи разрядов с 0 по $k - 1$. Десятичные числа строятся из 10 десятичных цифр:

0 1 2 3 4 5 6 7 8 9

Двоичные числа, напротив, строятся только из двух двоичных цифр:

0 1

Восьмеричные числа состоят из восьми цифр:

0 1 2 3 4 5 6 7

Для шестнадцатеричных чисел требуется 16 цифр. Это значит, что нам нужно 6 новых символов. Для обозначения цифр, следующих за 9, принято использовать прописные латинские буквы от A до F. Таким образом, шестнадцатеричные числа строятся из следующих цифр:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Двоичный разряд (то есть 1 или 0) обычно называют **битом**. На рис. А.2 десятичное число 2001 представлено в двоичной, восьмеричной и шестнадцатеричной системах счисления. Число 7B9, очевидно, шестнадцатеричное, поскольку символ B встречается только в шестнадцатеричных числах. А число 111 может быть записано в любой из четырех систем счисления. Чтобы избежать неоднозначности, следует указывать основание системы счисления (если оно не очевидно из контекста).

В табл. А.1 ряд неотрицательных целых чисел представлен в каждой из четырех систем счисления.

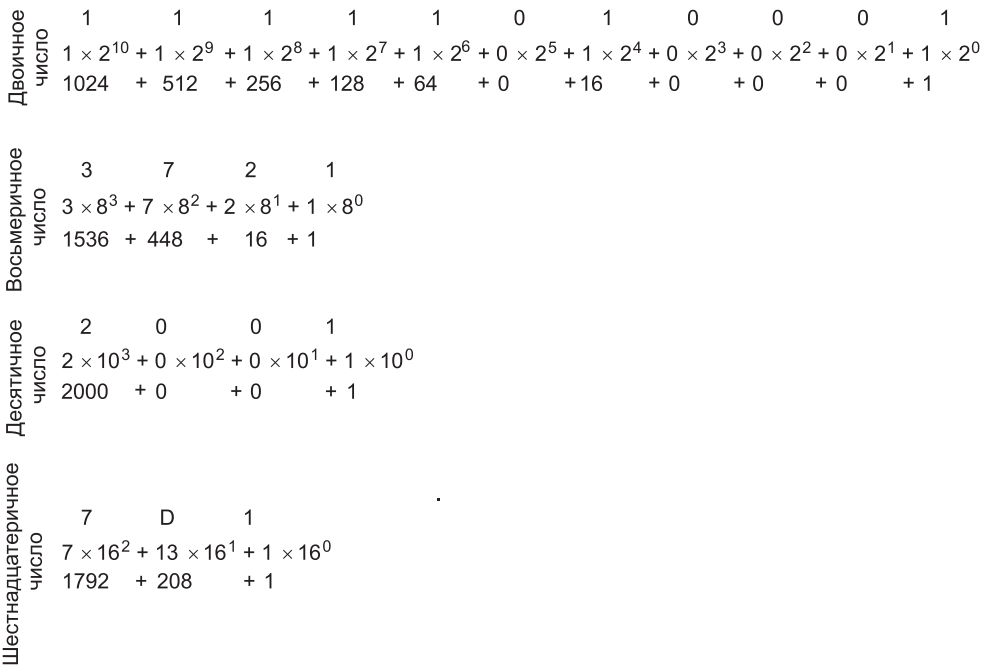


Рис. А.2. Число 2001 в двоичной, восьмеричной и шестнадцатеричной системах счисления

Таблица А.1. Десятичные числа и их двоичные, восьмеричные и шестнадцатеричные эквиваленты

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

продолжение

Таблица А.1 *(продолжение)*

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Преобразование чисел из одной системы счисления в другую

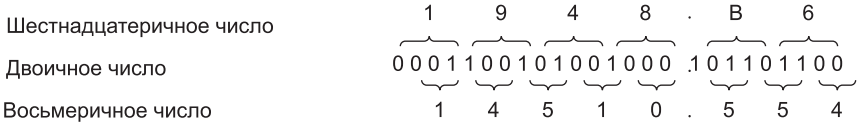
Преобразовывать числа из восьмеричной в шестнадцатеричную или в двоичную систему счисления и обратно легко. Чтобы преобразовать двоичное число в восьмеричное, нужно разделить его на группы по три бита, причем три бита непосредственно слева от двоичной запятой формируют одну группу, следующие три бита слева от этой группы формируют вторую группу и т. д. Каждую группу по три бита можно преобразовать в один восьмеричный разряд со значением от 0 до 7 (см. первые строки табл. А.1). Чтобы дополнить группу до трех битов, нужно спереди приписать один или два нуля. Преобразование из восьмеричной системы в двоичную тоже тривиально. Каждый восьмеричный разряд просто заменяется эквивалентным 3-разрядным числом. Преобразование из шестнадцатеричной в двоичную систему по сути сходно с преобразованием из восьмеричной в двоичную систему, только каждый шестнадцатеричный разряд соответствует группе из четырех битов, а не из трех. На рис. А.3 приведены примеры преобразований из одной системы в другую.

Преобразование десятичных чисел в двоичные можно совершать двумя разными способами. Первый способ непосредственно вытекает из определения двоичных чисел. Самая большая степень двойки, меньшая, чем число, вычитается из этого числа. Та же операция проделывается с полученной разностью. Когда число разложено по степеням двойки, двоичное число может быть получено следующим образом. Единички ставятся в тех позициях, которые соответствуют полученным степеням двойки, а нули — во всех остальных позициях.

Второй способ (подходящий только для целых чисел) — деление на 2. Частное записывается непосредственно под исходным числом, а остаток (0 или 1) записывается рядом с частным. То же проделывается с полученным частным. Процесс

повторяется до тех пор, пока не останется 0. В результате должно получиться две колонки чисел — частных и остатков. Двоичное число можно считать из колонки остатков снизу вверх. На рис. А.4 показано, как происходит преобразование из десятичной в двоичную систему.

Пример 1



Пример 2

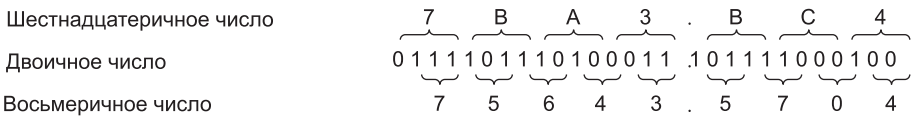


Рис. А.3. Примеры преобразования из восьмеричной системы счисления в двоичную и из шестнадцатеричной в двоичную

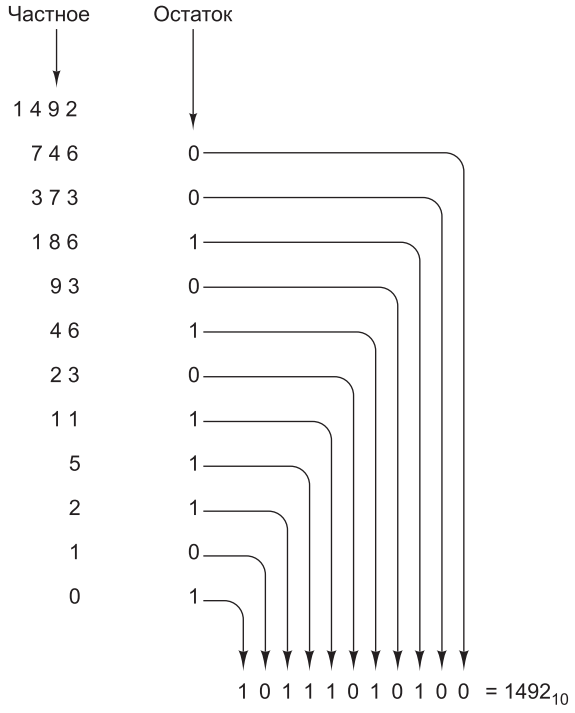


Рис. А.4. Преобразование десятичного числа 1492 в двоичное путем последовательного деления (сверху вниз). Например, 93 делится на 2, получается 46 и остаток 1. Остаток записывается в строку снизу

Двоичные числа можно преобразовывать в десятичные двумя способами. Первый способ — суммирование степеней двойки, которые соответствуют битам 1 в двоичном числе. Например:

$$10110 = 24 + 22 + 21 = 16 + 4 + 2 = 22.$$

Во втором способе двоичное число записывается вертикально по одному биту в строке, крайний левый бит находится внизу. Самая нижняя строка — это строка 1, затем идет строка 2 и т. д. Десятичное число строится напротив этой колонки. Сначала обозначим строку 1. Элемент строки n состоит из удвоенного элемента строки $n - 1$ плюс бит строки n (0 или 1). Элемент, полученный в самой верхней строке, и будет ответом. Метод иллюстрирует рис. А.5.

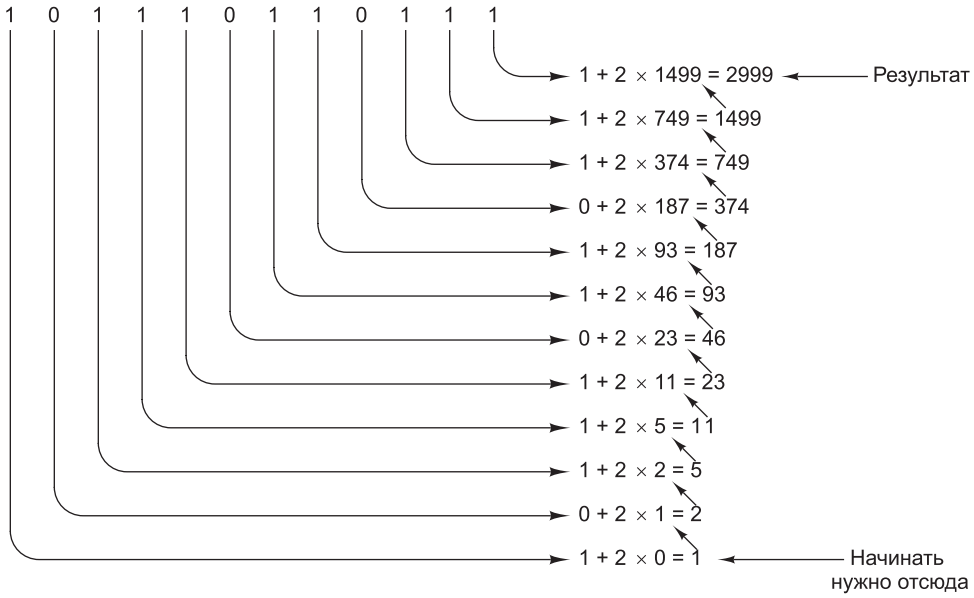


Рис. А.5. Преобразование двоичного числа 101110110111 в десятичное путем последовательного удваивания снизу вверх. В каждой следующей строке удваивается значение предыдущей строки и прибавляется соответствующий бит. Например, 374 умножается на 2 и прибавляется бит соответствующей строки (в данном случае 1). В результате получается 749

Преобразование из десятичной в восьмеричную или шестнадцатеричную систему можно выполнить либо путем преобразования сначала в двоичную, а затем в нужную нам систему, либо путем вычитания степеней 8 или 16.

Отрицательные двоичные числа

На протяжении всей истории цифровых компьютеров для представления отрицательных чисел использовались 4 различные системы. Первая из них называется системой **со знаком**. В такой системе крайний левый бит — это знаковый бит (0 — плюс, 1 — минус), а оставшиеся биты показывают абсолютное значение числа.

Во второй системе, которая называется **дополнением до единицы**, тоже присутствует знаковый бит (0 — плюс, 1 — минус). Чтобы сделать число отрицательным, нужно заменить каждую единицу нулем и каждый ноль единицей. Это относится и к знаковому биту. Система дополнения до единицы уже устарела.

Третья система, **дополнение до двух**, содержит знаковый бит (0 — плюс, 1 — минус). Отрицание числа происходит в два этапа. Сначала каждая единица меняется на ноль, а каждый ноль — на единицу (как и в системе дополнения до единицы). Затем к полученному результату прибавляется единица. Двоичное сложение происходит точно так же, как и десятичное, только перенос совершается в том случае, если сумма больше 1, а не больше 9. Например, рассмотрим преобразование числа 6 в форму с дополнением до двух:

- Число +6:

00000110.

- Число –6 в системе с дополнением до единицы:

11111001.

- Число –6 в системе с дополнением до двух:

11111010.

Если потребуется выполнить перенос из крайнего левого бита, он просто отбрасывается.

В четвертой системе, которая для m-разрядных чисел называется **системой со смещением на 2m–1**, число представляется как сумма этого числа и $2m–1$. Например, для 8-разрядного числа ($m = 8$) — это система со смещением на 128, в ней число сохраняется в виде суммы исходного числа и 128. Следовательно, –3 превращается в –3 + 128 = 125, и это число (–3) представляется 8-разрядным двоичным числом 125 (01111101). Числа от –128 до +127 выражаются числами от 0 до 255 (все их можно записать в виде 8-разрядного положительного числа). Отметим, что эта система соответствует системе с дополнением до двух с обращенным знаковым битом. В табл. А.2 представлены примеры отрицательных чисел во всех четырех системах.

Таблица А.2. Отрицательные 8-разрядные числа в четырех различных системах

N десятичное	N двоичное	–N в системе со знаком	–N дополнение до единицы	–N дополнение до двух	–N смещение на 128
1	00000001	10000001	11111110	11111111	011111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000

продолжение ⇩

Таблица А.2 *(продолжение)*

N десятичное	N двоичное	−N в системе со знаком	−N дополнение до единицы	−N дополнение до двух	−N смещение на 128
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	10101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Не существует	Не существует	Не существует	10000000	00000000

В системах со знаком и с дополнением до единицы есть два представления нуля: +0 и –0. Такая ситуация нежелательна. В системе с дополнением до двух такой проблемы нет, поскольку здесь плюс ноль — это всегда плюс ноль. Но зато в этой системе есть другая особенность. Набор битов, состоящий из единицы, за которой следуют все нули, является дополнением самого себя. В результате ряд положительных и отрицательных чисел несимметричен — существует одно отрицательное число без соответствующего ему положительного.

Мы считаем, что это проблема, поскольку хотим иметь систему кодировки, в которой:

- существует только одно представление нуля;
- количество положительных чисел равно количеству отрицательных.

Дело в том, что любой ряд чисел с равным количеством положительных и отрицательных чисел и только одним нулем содержит нечетное число членов, тогда как *m* бит предполагают четное число битовых комбинаций. В любом случае либо одна битовая комбинация окажется лишней, либо одной комбинации будет недоставать. Лишнюю битовую комбинацию можно использовать для обозначения числа –0, для большого отрицательного числа или для чего-нибудь еще, но она всегда будет создавать неудобства.

Двоичная арифметика

Рисунок А.6 иллюстрирует сложение двоичных чисел.

Сложение двух двоичных чисел начинается с крайнего правого бита. Суммируются соответствующие биты в первом и втором слагаемых. Перенос совершается на одну позицию влево, как и в десятичной арифметике. В арифметике

Первое слагаемое	0	0	1	1
Второе слагаемое	+0	+1	+0	+1
Сумма	0	1	1	0
Перенос	0	0	0	1

Рис. А.6. Таблица сложения для двоичных чисел

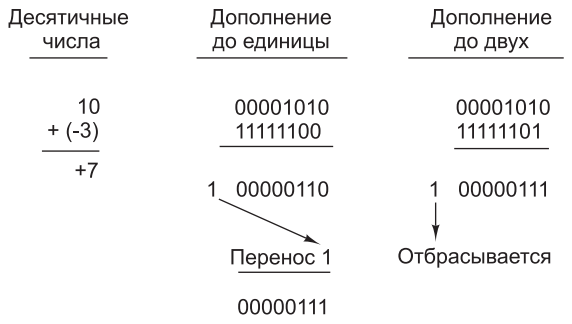


Рис. А.7. Сложение в системах с дополнением до единицы и с дополнением до двух

с дополнением до единицы бит переноса после сложения крайних левых битов прибавляется к крайнему правому биту. Этот процесс называется циклическим переносом. В арифметике с дополнением до двух бит переноса, полученный в результате сложения крайних левых битов, просто отбрасывается. Примеры арифметических действий над двоичными числами показаны на рис. А.7.

Если первое и второе слагаемые имеют противоположные знаки, ошибки переполнения не произойдет. Если они имеют одинаковые знаки, а результат — противоположный знак, значит, произошла ошибка переполнения и результат неверен. И в арифметике с дополнением до единицы, и в арифметике с дополнением до двух переполнение происходит тогда и только тогда, когда перенос в знаковый бит отличается от переноса из знакового бита. В большинстве компьютеров перенос из знакового бита сохраняется, но перенос в знаковый бит из результата не виден, поэтому обычно вводится специальный бит переполнения.

Вопросы и задания

1. Преобразуйте следующие числа в двоичные: 1984, 4000, 8192.
2. Преобразуйте двоичное число 1001101001 в десятичную, восьмеричную и шестнадцатеричную системы.
3. Какие из символьных строк BED, CAB, DEAD, DECADE, ACCEDED, BAG, DAD являются шестнадцатеричными числами?
4. Выразите десятичное число 100 в системах счисления с основаниями от 2 до 9.
5. Сколько различных положительных целых чисел можно выразить в k разрядах, используя числа с основанием системы счисления r?

6. Большинство людей с помощью пальцев на руках могут сосчитать до 10. Однако компьютерщики способны на большее. Представим, что каждый палец соответствует одному двоичному разряду. Пусть вытянутый палец означает 1, а загнутый — 0. До скольки мы можем сосчитать, используя пальцы обеих рук? А если рассматривать пальцы на руках и на ногах? Представим, что большой палец левой ноги — это знаковый бит для чисел с дополнением до двух. Сколько чисел можно выразить таким способом?
7. Выполните следующие вычисления над 8-разрядными числами с дополнением до двух:

   00101101 + 01101111

   11111111 + 11111111

   00000000 – 11111111

   11110111 – 11110111

8. Выполните те же вычисления в системе с дополнением до единицы.
9. Далее приведены задания на сложение 3-разрядных двоичных чисел в системе с дополнением до двух. Для каждой суммы установите:
   1) Равен ли знаковый бит результата единице.
   2) Равны ли младшие три бита нулю.
   3) Не произошло ли переполнения.

   000 + 001

   000 + 111

   111 + 110

   100 + 111

   100 + 100

10. Десятичные числа со знаком, состоящие из n разрядов, можно представить в $n + 1$ разрядах без знака. Положительные числа содержат 0 в крайнем левом разряде. Отрицательные числа получаются путем вычитания каждого разряда из 9. Например, отрицательным числом для 014 725 будет 985 274. Такие числа называются числами с дополнением до девяти. Они аналогичны двоичным числам с дополнением до единицы. Выразите следующие числа в виде 3-разрядных чисел в системе с дополнением до девяти: 6, –2, 100, –14, –1, 0.
11. Сформулируйте правило для сложения чисел с дополнением до девяти, а затем выполните следующие вычисления:

    0001 + 9999

    0001 + 9998

    9997 +9996

    9241 + 0802

12. Система с дополнением до десяти аналогична системе с дополнением до двух. Отрицательное число в системе с дополнением до десяти получается путем прибавления единицы к соответствующему числу с дополнением до девяти без учета переноса. По какому правилу происходит сложение в системе с дополнением до десяти?
13. Составьте таблицы умножения для чисел системы счисления с основанием 3.
14. Перемножьте двоичные числа 0111 и 0011.
15. Напишите программу, которая на входе получает десятичное число со знаком в виде строки ASCII-символов, а на выходе выводит представление этого числа в восьмеричной и шестнадцатеричной системах, а также в двоичной системе с дополнением до двух.
16. Напишите программу, которая на входе получает две строки из 32 ASCII-символов нулей и единиц. Каждая строка представляет 32-разрядное двоичное число в системе с дополнением до двух. На выходе программа должна выдавать их сумму в виде строки из 32 ASCII-символов нулей и единиц.

Приложение Б
Числа с плавающей точкой

Диапазон чисел, используемых при различных вычислениях, очень велик. Например, в астрономические вычисления может включаться масса электрона (9×10^{-28} граммов) и масса солнца (2×10^{33} граммов). Диапазон чисел здесь превышает 10^{60}. Эти числа можно представить следующим образом:

000000000000000000000000000000000.00000000000000000000000000009

200000000000000000000000000000000.00000000000000000000000000000

При всех вычислениях должны сохраняться 34 разряда слева от десятичной точки и 28 разрядов справа от нее. Это даст 62 значимых разряда в результатах. На бинарном компьютере можно использовать арифметику с многократно увеличенной точностью, чтобы обеспечить достаточную значимость. Однако мы не можем определить массу солнца с точностью даже до пяти значимых разрядов, не говоря уже о 62. В действительности практически невозможно выполнить какие-либо измерения с точностью до 62 знаков. Можно было бы хранить все промежуточные результаты с точностью до 62 значимых разрядов, а перед выводом окончательных результатов отбрасывать 50 или 60 разрядов, но процессор и память тратили бы на это слишком много времени.

Нам нужна такая система для представления чисел, в которой диапазон выражаемых чисел не зависит от числа значимых разрядов. В этом приложении мы расскажем о такой системе. В ее основе лежит экспоненциальное представление чисел, которое применяется в физике, химии и машиностроении.

Принципы представления чисел с плавающей точкой

Числа можно выражать в следующей общепринятой экспоненциальной форме:

$$n = f \times 10^{e},$$

где f называется **мантиссой**, а e (это положительное или отрицательное целое число) — **экспонентой**. Компьютерная версия такого представления называется представлением числа с **плавающей точкой**. Вот примеры чисел в такой записи:

$$3{,}14 = 0{,}314 \times 10^{1} = 3{,}14 \times 100.$$

$$0{,}000001 = 0{,}1 \times 10^{-5} = 1{,}0 \times 10^{-6}.$$

$$1941 = 0{,}1941 \times 10^{4} = 1{,}94^{1} \times 10^{3}.$$

Область значений определяется по числу разрядов в экспоненте, а точность — по числу разрядов в мантиссе. Существуют несколько способов представления того или иного числа, поэтому одна форма выбирается в качестве стандартной.

Чтобы изучить свойства такого способа представления, рассмотрим представление R с трехразрядной мантиссой со знаком в диапазоне $0{,}1 \le |f| < 1$ и двухразрядной экспонентой со знаком. Эти числа находятся в диапазоне от $+0{,}100 \times \times 10{-}99$ до $+0{,}999 \times 10{+}99$, то есть простираются почти на 199 значимых разрядов, хотя для записи числа требуется всего 5 разрядов и 2 знака.

Числа с плавающей точкой можно использовать для моделирования системы действительных чисел в математике, хотя здесь есть несколько существенных различий. На рис. Б.1 представлена ось действительных чисел. Она разбита на 7 областей:

1. Отрицательные числа меньше $-0{,}999 \times 1099$.
2. Отрицательные числа от $-0{,}999 \times 1099$ до $-0{,}100 \times 10{-}99$.
3. Отрицательные числа от $-0{,}100 \times 10{-}99$ до нуля.
4. Нуль.
5. Положительные числа от 0 до $0{,}100 \times 10{-}99$.
6. Положительные числа от $0{,}100 \times 10{-}99$ до $0{,}999 \times 1099$.
7. Положительные числа больше $0{,}999 \times 1099$.

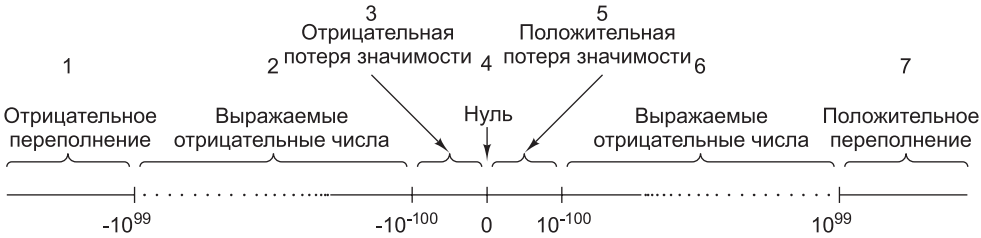


Рис. Б.1. Ось действительных чисел разбита на 7 областей

Первое отличие действительных чисел от чисел с плавающей точкой, которые записываются тремя разрядами в мантиссе и двумя разрядами в экспоненте, состоит в том, что последние нельзя использовать для записи чисел из областей 1, 3, 5 и 7. Если в результате арифметической операции получится число из области 1 или 7 (например, $1060 \times 1060 = 10120$), то произойдет **ошибка переполнения**, и результат будет неверным. Причина — ограничение области значений чисел в данном представлении. Точно так же нельзя выразить результат из области 3 или 5. Такая ситуация называется **ошибкой потери значимости**. Эта ошибка менее серьезна, чем ошибка переполнения, поскольку часто нуль является вполне удовлетворительным приближением для чисел из областей 3 или 5. Остаток счета в банке на 10–102 не сильно отличается от нулевого остатка счета.

Второе важное отличие чисел с плавающей точкой от действительных чисел — их плотность. Между любыми двумя действительными числами x и y существует другое действительное число независимо от того, насколько близко к y расположено число x. Это свойство вытекает из того, что для любых отличных действительных чисел x и y между ними существует действительное число $z = (x + y)/2$. Действительные числа формируют континуум.

Числа с плавающей точкой континуума не формируют. В двухзнаковой пятиразрядной системе можно выразить ровно 179 100 положительных чисел, 179 100 отрицательных чисел и 0 (который можно выразить разными способами), то есть

всего 358 201 чисел. Из бесконечного числа действительных чисел в диапазоне от −10+100 до +0,999 × 1099 в этой системе можно выразить только 358 201 число. На рис. Б.1 эти числа показаны точками. Результат вычислений может быть и другим числом, даже если он находится в области 2 или 6. Например, результат деления числа +0,100 × 103 на 3 нельзя выразить *точно* в нашем представлении. Если полученное число нельзя выразить с помощью используемого представления, нужно брать ближайшее представимое число. Такой процесс называется **округлением**.

Промежутки между смежными числами, которые можно выразить в представлении с плавающей точкой, в областях 2 и 6 не постоянны. Промежуток между числами +0,998 × 1099 и +0,999 × 1099 гораздо больше промежутка между числами +0,998 × 100 и +0,999 × 100. Однако если промежутки между числом и его соседом выразить как процентное отношение от этого числа, большой разницы в промежутках не будет. Другими словами, **относительная погрешность**, полученная при округлении, приблизительно равна и для малых, и для больших чисел.

Выводы, сделанные для системы представления с трехразрядной мантиссой и двухразрядной экспонентой, справедливы и для других представлений чисел. При изменении числа разрядов в мантиссе или экспоненте просто сдвигаются границы областей 2 и 6 и меняется число представляемых единиц в этих областях. С увеличением числа разрядов в мантиссе увеличивается плотность элементов и, следовательно, точность приближения. С увеличением количества разрядов в экспоненте области 2 и 6 увеличиваются за счет уменьшения областей 1, 3, 5 и 7. В табл. Б.1 показаны приблизительные границы области 6 для десятичных чисел с плавающей точкой и различным количеством разрядов в мантиссе и экспоненте.

Таблица Б.1. Приблизительные верхняя и нижняя границы чисел с плавающей точкой

Количество разрядов в мантиссе	Количество разрядов в экспоненте	Нижняя граница	Верхняя граница
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Вариант такого представления применяется в компьютерах. Основа возведения в степень — 2, 4, 8 или 16, но не 10. В этом случае мантисса состоит из цепочки двоичных, четверичных, восьмеричных и шестнадцатеричных разрядов. Если крайний левый разряд равен 0, то все разряды можно сместить на один влево, а экспоненту уменьшить на 1, не меняя при этом значения числа (исключение составляет ситуация потери значимости). Мантисса с ненулевым крайним левым разрядом называется **нормализованной**.

Нормализованные числа обычно предпочитаются ненормализованным, поскольку существует только одна нормализованная форма, а ненормализованных форм может быть много. Примеры нормализованных чисел с плавающей запятой даны на рис. Б.2 для двух оснований степени. В этих примерах показана 16-разрядная мантисса (включая знаковый бит) и 7-разрядная экспонента. Точка находится слева от крайнего левого бита мантиссы и справа от экспоненты.

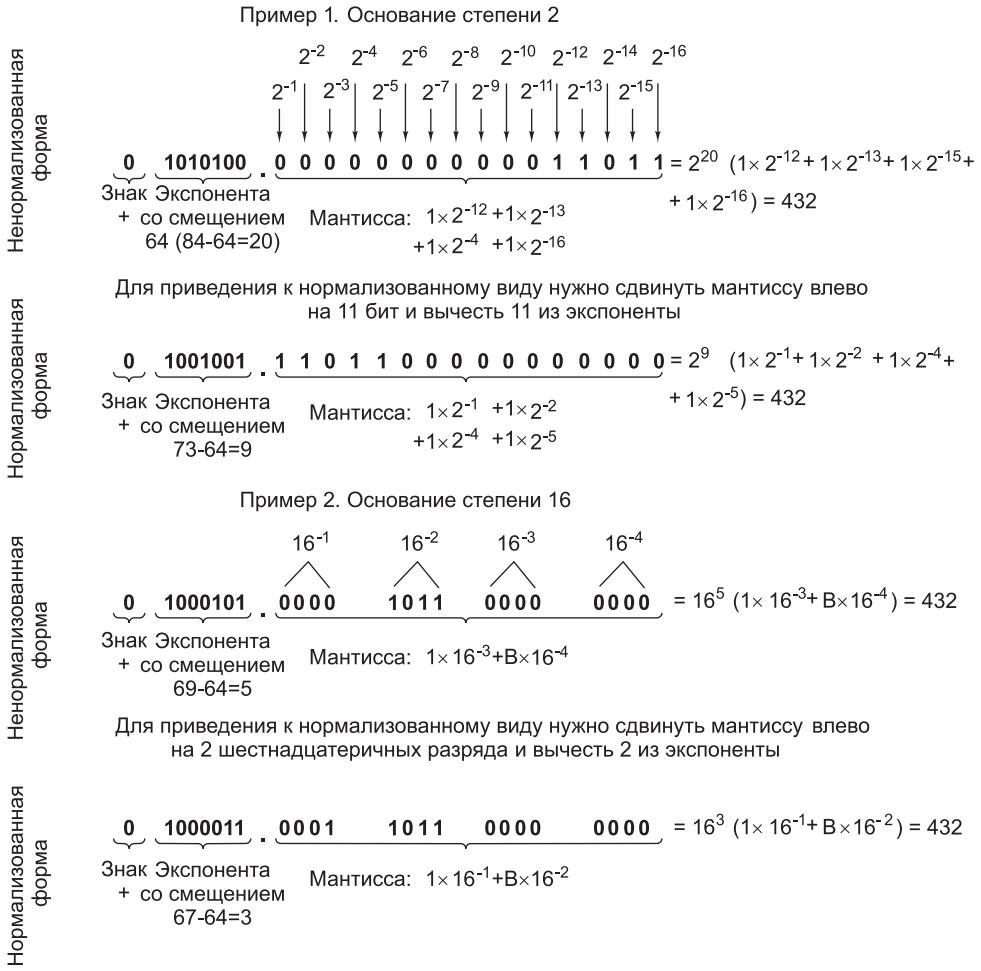


Рис. Б.2. Примеры нормализованных чисел с плавающей точкой

Стандарт IEEE 754

До 80-х годов каждый производитель поддерживал собственный формат чисел с плавающей точкой. Все они отличались друг от друга. Более того, в некоторых из них арифметические действия выполнялись неправильно, поскольку арифметика с плавающей точкой имеет некоторые тонкости, которые не очевидны для среднестатистического разработчика аппаратного обеспечения.

Чтобы изменить эту ситуацию, в конце 70-х годов институт IEEE учредил комиссию по стандартизации арифметики с плавающей точкой. Целью было не только дать возможность переносить данные с одного компьютера на другой, но и обеспечить разработчиков аппаратного обеспечения заведомо правильной моделью. В результате в 1985 году вышел стандарт IEEE 754 [IEEE, 1985]. В настоящее время большинство процессоров (в том числе Intel, SPARC и JVM) содержат команды с плавающей точкой, которые соответствуют этому стандарту. В отличие от многих стандартов, ставших плодом неудачных компромиссов и мало кого устраивавших, этот стандарт неплох, причем в значительной степени благодаря тому, что его изначально разрабатывал один человек, профессор математики университета Беркли Вильям Каган (William Kahan). Рассмотрим этот стандарт.

Стандарт IEEE 754 определяет три формата: с одинарной точностью (32 бита), с удвоенной точностью (64 бита) и с повышенной точностью (80 бит). Формат с повышенной точностью предназначен для уменьшения ошибки округления. Он применяется главным образом в арифметических устройствах с плавающей точкой, поэтому мы не будем о нем говорить. В форматах с одинарной и удвоенной точностью используются основание степени 2 для мантисс и смещенная экспонента. Форматы представлены на рис. Б.3.

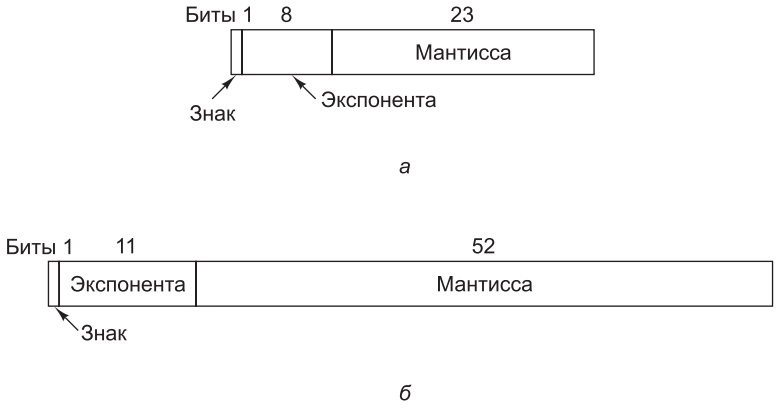


Рис. Б.3. Форматы стандарта IEEE с плавающей запятой: одинарная точность (*а*); удвоенная точность (*б*)

Оба формата начинаются со знакового бита для всего числа; 0 указывает на положительное число, 1 — на отрицательное. Затем следует смещенная экспонента. Для формата одинарной точности смещение равно 127, а для формата удвоенной точности — 1023. Минимальная (0) и максимальная (255 и 2047) экс-

поненты не используются для нормализованных чисел. У них есть специальное предназначение, о котором мы поговорим позже. В конце идут мантиссы по 23 и 52 бита соответственно.

Нормализованная мантисса начинается с двоичной точки, за которой следует 1 бит, а затем — остаток мантиссы. Следуя практике, начатой с компьютера PDP-11, компьютерщики осознали, что 1 бит перед мантиссой сохранять не нужно, а просто считать, что он там есть. Следовательно, стандарт определяет мантиссу следующим образом. Она состоит из неявного бита, который всегда равен 1, неявной двоичной точке, за которыми идут 23 или 52 произвольных бита. Если все 23 или 52 бита мантиссы равны 0, то мантисса имеет значение 1,0. Если все биты мантиссы равны 1, то числовое значение мантиссы немного меньше, чем 2,0. Во избежание путаницы в английском языке для обозначения комбинации из неявного бита, неявной двоичной точки и 23 или 52 явных битов вместо термина **мантисса** (mantissa) используется термин **значащая часть числа** (significand). Значащая часть числа (s) всех нормализованных чисел лежит в диапазоне $1 \le s < 2$.

Числовые характеристики стандарта IEEE для чисел с плавающей точкой даны в табл. Б.2. В качестве примеров рассмотрим числа 0,5, 1 и 1,5 в нормализованном формате с одинарной точностью. Они представлены шестнадцатеричными числами 3F000000, 3F800000 и 3FC00000 соответственно.

Таблица Б.2. Характеристики чисел с плавающей точкой стандарта IEEE

	Одинарная точность	**Удвоенная точность**
Количество битов в знаке	1	1
Количество битов в экспоненте	8	11
Количество битов в мантиссе	23	52
Общее число битов	32	64
Смещение экспоненты	Смещение 127	Смещение 1023
Область значений экспоненты	От –126 до +127	от –1022 до +1023
Самое маленькое нормализованное число	2^{-126}	2^{-1022}
Самое большое нормализованное число	Приблизительно 2^{128}	Приблизительно 2^{1024}
Диапазон десятичных дробей	Приблизительно от 10^{-38} до 10^{38}	Приблизительно от 10^{-308} до 10^{308}
Самое маленькое ненормализованное число	Приблизительно 10^{-45}	Приблизительно 10^{-324}

Традиционные проблемы, связанные с числами с плавающей точкой, — переполнение, потеря значимости и неинициализированные числа. Подход, используемый в стандарте IEEE, отчасти заимствован у машины CDC 6600. Помимо нормализованных чисел в стандарте предусмотрено еще 4 типа чисел (рис. Б.4).

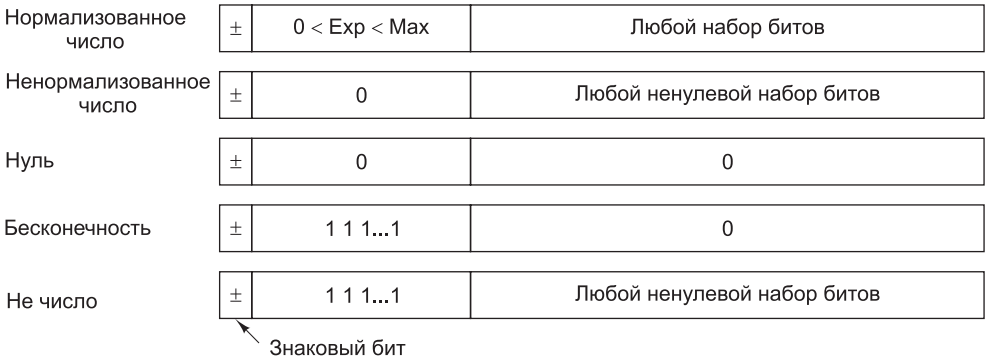


Рис. Б.4. Числовые типы стандарта IEEE

Проблема возникает в том случае, если абсолютное значение (модуль) результата меньше самого маленького нормализованного числа с плавающей точкой, которое можно представить в этой системе. Раньше аппаратное обеспечение действовало одним из двух способов: либо устанавливало результат на 0, либо вызывало ошибку потери значимости. Ни один из этих двух способов не является удовлетворительным, поэтому в стандарт IEEE введены **ненормализованные числа**. Эти числа имеют экспоненту 0 и мантиссу, представленную следующими 23 или 52 битами. Неявный бит 1 слева от двоичной точки превращается в 0. Ненормализованные числа можно легко отличить от нормализованных, поскольку у последних не может быть нулевой экспоненты.

Самое маленькое нормализованное число с одинарной точностью содержит 1 в экспоненте и 0 в мантиссе и представляет $1{,}0 \times 2^{-126}$. Самое большое ненормализованное число содержит 0 в экспоненте и все единицы в мантиссе и представляет примерно $0{,}9999999 \times 2^{-127}$, то есть почти то же самое число. Следует отметить, что это число содержит только 23 бита значимости, а все нормализованные числа — 24 бита.

По мере уменьшения результата при дальнейших вычислениях экспонента попрежнему остается равной 0, а первые несколько битов мантиссы превращаются в нули, что уменьшает и значение, и число значимых битов мантиссы. Самое маленькое ненулевое ненормализованное число содержит 1 в крайнем правом бите, а все остальные биты равны 0. Экспонента представляет 2^{-127}, а мантисса — 2^{-23}, поэтому значение равно 2^{-150}. Такая схема предусматривает постепенное исчезновение значимых разрядов, а не перескакивает на 0, когда результат не удается выразить в виде нормализованного числа.

В этой схеме присутствуют два нуля, положительный и отрицательный, определяемые по знаковому биту. Оба имеют экспоненту 0 и мантиссу 0. Здесь тоже бит слева от двоичной точки по умолчанию равен 0, а не 1.

Простого решения проблемы переполнения нет. Для этого существует специальное представление бесконечности: с экспонентой, содержащей все единицы, и мантиссой, равной 0. Это число можно использовать в качестве операнда. Оно подчиняется обычным математическим правилам для бесконечности. Например, бесконечность и любое число в сумме дают бесконечность. Конечное число, деленное на бесконечность, равно 0. Любое конечное число, разделенное на 0, стремится к бесконечности.

А что получится, если бесконечность разделить на бесконечность? Результат не определен. Для такого случая существует другой специальный формат — **не число** (Not a Number, **NaN**). Его тоже можно использовать в качестве операнда.

Вопросы и задания

1. Преобразуйте следующие числа в формат стандарта IEEE с одинарной точностью. Результаты представьте в восьми шестнадцатеричных разрядах.
   1) 9;
   2) 5/32;
   3) –5/32;
   4) 6,125.
2. Преобразуйте следующие числа с плавающей точкой одинарной точности из шестнадцатеричной в десятичную систему счисления:
   1) 42E28000H;
   2) 3F880000H;
   3) 00800000H;
   4) C7F00000H.
3. Числа с плавающей точкой в формате одинарной точности в IBM/370 состоят из 7-разрядной смещенной экспоненты (смещение равно 64), 24-разрядной мантиссы и знакового бита. Двоичная точка находится слева от мантиссы. Основание степени — 16. Порядок полей — знаковый бит, экспонента, мантисса. Выразите число 7/64 в виде нормализованного шестнадцатеричного числа в этой системе.
4. Следующие двоичные числа с плавающей точкой состоят из знакового бита, смещенной экспоненты (смещение равно 64) с основанием 2 и 16-разрядной мантиссы. Нормализуйте их:
   1) 0 1000000 0001010100000001;
   2) 0 0111111 0000001111111111;
   3) 0 1000011 1000000000000000.
5. Чтобы сложить два числа с плавающей точкой, нужно уровнять экспоненты (сдвинув мантиссу). Затем можно сложить мантиссы и нормализовать результат, если в этом есть необходимость. Сложите числа одинарной точности 3EE00000H и 3D800000H и выразите нормализованный результат в шестнадцатеричной системе счисления.
6. Компьютерная компания решила выпустить машину, поддерживающую 16-разрядные числа с плавающей точкой. В модели 0.001 формат состоит из знакового бита, 7-разрядной смещенной экспоненты (смещение равно 64) и 8-разрядной мантиссы. В модели 0.002 формат состоит из знакового бита, 5-разрядной смещенной экспоненты (смещение равно 16) и 10-разрядной мантиссы. В обеих моделях основание степени равно 2. Каково самое маленькое и самое большое положительное нормализованное число в этих моделях?

Сколько десятичных разрядов точности содержится в каждой модели? А вы купили бы какую-нибудь из этих двух моделей?

7. Существует одна ситуация, при которой операция над двумя числами с плавающей точкой может вызвать радикальное сокращение количества значимых битов в результате. Что это за ситуация?
8. Некоторые микросхемы для обработки команд с плавающей точкой имеют встроенную команду извлечения квадратного корня. Возможно применение итерационного алгоритма (например, метода Ньютона–Рафсона). Итерационные алгоритмы дают последовательные приближения решения. Как можно быстро получить приближенный квадратный корень от числа с плавающей точкой?
9. Напишите процедуру сложения двух чисел одинарной точности с плавающей точкой. Каждое число представлено 32-элементным логическим массивом.
10. Напишите процедуру сложения двух чисел с плавающей точкой одинарной точности, в которых для экспоненты используется основание системы счисления 16, а для мантиссы — основание системы счисления 2, кроме того, числа не содержат неявного бита 1 слева от двоичной точки. В нормализованном числе крайние левые 4 бита мантиссы могут быть равны 0001, 0010, ..., 1111, но не 0000. Число нормализуется путем сдвига мантиссы влево на 4 бита и прибавления 1 к экспоненте.

Приложение B Программирование на языке ассемблера

(Эверт Ваттель, Свободный университет Амстердама, Голландия)

В каждом компьютере есть **уровень архитектуры набора команд (**Instruction Set Architecture, **ISA)**, который представляет собой совокупность регистров, команд и других элементов, доступных программистам, пишущим на языках низкого уровня. Уровень ISA часто называют **машинным языком**, но это не совсем точное определение. Программа, написанная на машинном языке, являет собой длинный список двоичных чисел, описывающих исполняемые команды и их операнды, в котором одно число соответствует одной команде. Писать программы с операторами в виде двоичных чисел очень сложно, поэтому во всех машинах предусмотрен **язык ассемблера** — символическое представление архитектуры набора команд, в котором двоичные числа заменены именами наподобие `ADD`, `SUB` и `MUL`. В этом приложении содержится краткое руководство по программированию на языке ассемблера процессора 8088 компании Intel. Этот язык был реализован в первых машинах IBM PC и именно на его основе развился современный язык ассемблера Core i7. Помимо прочего, в приложении описываются некоторые инструментальные средства, доступные в Интернете и способные помочь в процессе обучения программированию на языке ассемблера.

Назначение этого приложения не в том, чтобы готовить профессиональных программистов, пишущих на ассемблере, а в том, чтобы помочь читателю освоить архитектуру вычислительных систем через практические примеры. Именно по этой причине в качестве основы для изложения материала выбрана такая простая машина, как 8088 производства Intel. Сами по себе модели 8088 встречаются сейчас очень редко, но программы, написанные для них, можно исполнять и на современных системах Core i7, что позволяет читателю воочию наблюдать результаты своей работы. Скажем больше — значительная часть команд ядра Core i7 совпадает с аналогичными командами 8088; различие между ними сводится лишь к разрядности регистров (32 и 16 бит соответственно). Таким образом, это приложение можно рассматривать как введение в курс программирования на языке ассемблера Core i7.

Чтобы программировать любую машину на ее ассемблере, программист должен хорошо разбираться в уровне архитектуры набора команд такой машины. По этой причине разделы «Основные понятия», «Процессор 8088», «Память и адресация» и «Набор команд 8088» посвящены детальному рассмотрению архитектуры, организации памяти, режимам адресации и командам 8088. В разделе «Ассемблер» содержится описание программного обеспечения, предназначенного для программирования на ассемблере. К этому программному

обеспечению, которое распространяется совершенно бесплатно, мы будем обращаться впоследствии при изучении примеров программ. Следует учесть, что в ассемблерах других процессоров могут применяться оригинальные нотации, и к этому нужно быть готовым. В разделе «Трассер» рассматриваются программные средства интерпретации (трассировки, отладки), помогающие начинающим программистам устранять ошибки в программах. Раздел «Подготовительные действия» посвящен установке программного обеспечения и базовым приемам работы с ним. Наконец, в разделе «Примеры» содержатся готовые программы, примеры, задачи и их решения.

Основные понятия

Наш обзор программирования на ассемблере мы начнем с краткого описания самого языка ассемблера, после чего приведем небольшой пример.

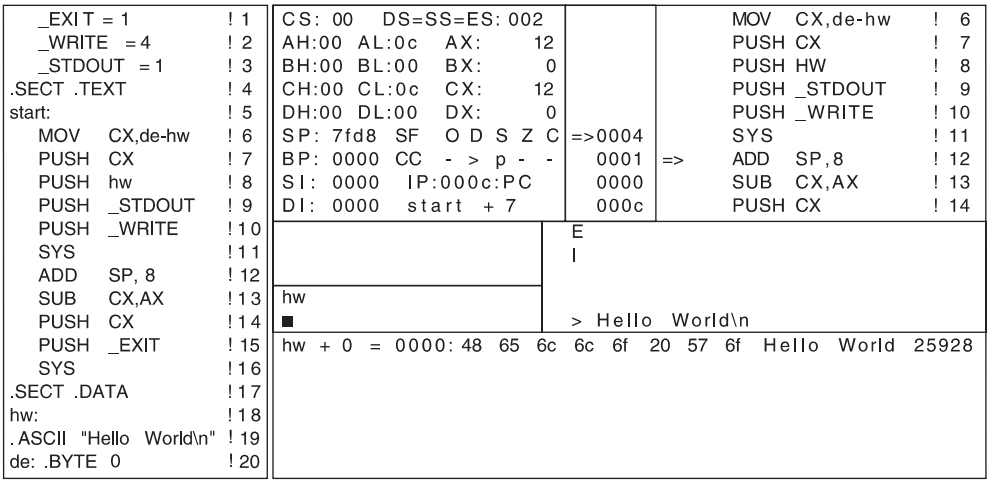
Язык ассемблера

Каждый язык ассемблера, или просто ассемблер, основывается на **мнемониках** — кратких осмысленных с точки зрения программиста обозначениях машинных команд (сложения, вычитания, умножения и других) наподобие `ADD`, `SUB` и `MUL`. Кроме того, ассемблеры допускают применение **символических имен** для обозначения констант и **меток** для обозначения команд и адресов памяти. Кроме того, большинство языков ассемблера поддерживают некоторое количество **псевдокоманд**, которые не относятся к уровню архитектуры набора команд, но регулируют процесс ассемблирования.

При ассемблировании программы, написанной на языке ассемблера, с помощью специальной программы (она так и называется — **ассемблер**) получается **двоичная программа**, готовая непосредственно к исполнению в реальной аппаратной среде. Здесь следует отметить, что начинающие программисты, составляющие текст программы на языке ассемблера, часто допускают ошибки, при обнаружении которых исполнение двоичной программы останавливается без указания причин. Чтобы упросить задачу программиста, двоичную программу можно запустить не в реальной аппаратной среде, а в симуляторе, который в каждый отдельный момент времени исполняет только одну команду и выводит детальный отчет о своих действиях. В этом случае отлаживать программы становится значительно проще. В среде симулятора программы исполняются значительно медленнее, чем в реальных условиях, но цель — научиться программировать на языке ассемблера — в данном случае оправдывает средства. При дальнейшем изложении материала мы подразумеваем применение такого симулятора (называемого также **интерпретатором**, или **трассером**), который интерпретирует и отслеживает исполнение каждой команды двоичной программы. Термины «симулятор», «интерпретатор» и «трассер», таким образом, употребляются в нашем тексте в одинаковом значении. Как правило, говоря об исполнении программы в среде симулятора, эту среду называют «интерпретатором»; если же речь идет о ее применение в качестве инструмента отладки, применяется термин «трассер»; в любом случае, имеется в виду одна и та же программа.

Небольшая программа на языке ассемблера

Переходя от теории к практике, рассмотрим короткую программу на языке ассемблера и вывод ее текста в интерпретаторе. На рис. В.1, *а* представлена программа на языке ассемблера 8088. Числа, следующие за восклицательными знаками, обозначают номера строк; они введены с единственной целью — упростить построчный анализ программы. Текст этой программы также размещен на сопроводительном компакт-диске (файл HlloWrld.s) в папке examples. Расширение этой программы (s), как и всех других, к которым мы будем обращаться, указывает на то, что она написана на языке ассемблера и не преобразована в двоичную форму. В окне трассера, показанном на рис. В.1, *б*, располагаются семь дочерних окон. В каждом из них указываются те или иные сведения о состоянии исполняемой двоичной программы.



а *б*

Рис. В.1. Исходный текст программы на языке ассемблера (*а*); окно трассера с информацией о ходе исполнения программы (*б*)

Теперь вкратце рассмотрим содержимое семи дочерних окон, изображенных на рис. В.1, *б*. В верхнем ряду расположены три окна — два больших и одно поменьше. В верхнем левом окне показано содержимое процессора, а именно — текущие значения сегментных (CS, DS, SS и ES), арифметических (AH, AL, AX) и других регистров.

В среднем окне в верхнем ряду указывается содержимое стека — области памяти, предназначенной для хранения временных значений.

В верхнем правом окне выводится фрагмент программы на языке ассемблера; стрелка указывает на команду, которая исполняется в данный момент. По мере исполнения программы эта стрелка, естественно, перемещается от одной команды к другой. Интерпретатор может работать в таком режиме, при котором однократное нажатие клавиши Enter приводит к исполнению одной команды и соответствующему обновлению всех окон. Таким образом, программу, написанную на языке ассемблера, в среде симулятора можно исполнять с той скоростью, которая позволяет разобраться в происходящем.

Окно, находящееся под верхним левым окном, определяет содержимое стека вызова подпрограмм, который в данном случае пуст. Ниже размещаются собственные команды трассера. Окно, расположенное справа от этих двух окон, предназначено для входных и выходных сообщений, а также сообщений об ошибках. В самом нижнем окне выводится содержимое части памяти.

Подробно обо всех этих окнах мы поговорим позже, а на данный момент важно получить общее представление о том, какие сведения трассер позволяет увидеть пользователю: исходный текст программы, содержимое регистров машины, информацию о состоянии исполняемой программы. При запуске каждой последующей команды содержимое окна интерпретатора обновляется, поэтому программист может разобраться в процессе настолько подробно, насколько он захочет.

Процессор 8088

Любой процессор, в том числе и 8088, обладает своим внутреннем состоянием, под которым понимается та или иная критически важная информация. Для хранения и обработки этой информации в процессоре предусмотрен особый набор **регистров**. Наиболее важным из них является регистр PC (Program Counter — **счетчик команд**). В нем указывается ячейка памяти (**адрес**), в которой хранится следующая в порядке исполнения команда. Другое обозначение этого регистра — IP (Instruction Pointer — **указатель команд**). Та часть основной памяти, в которой хранится следующая команда в порядке исполнения, называется **кодовым сегментом**. Допустимая емкость основной памяти, которую поддерживает процессор 8088, составляет чуть более 1 Мбайт, однако размер кодового сегмента ограничен значением 64 Кбайт. Начало этого сегмента в рамках памяти емкостью 1 Мбайт определяется в регистре CS (см. рис. В.1). Чтобы активировать новый кодовый сегмент, достаточно изменить значение регистра CS. Помимо кодового сегмента, существует сегмент данных, также занимающий 64 Кбайт и определяющий начало данных. На рис. В.1 начало данных определяется регистром DS, содержание которого можно при необходимости изменить и тем самым получить доступ к данным, находящимся за пределами текущего сегмента. Потребность в регистрах CS и DS обусловлена тем, что разрядность регистров 8088 составляет 16 бит, а, значит, хранить в них 20-разрядные адреса, необходимые для обращения к памяти емкостью 1 Мбайт, невозможно. Вот почему появились кодовый сегмент и сегмент данных.

В других регистрах содержатся данные или указатели на данные, размещенные в основной памяти. Ассемблерные программы обращаются к этим регистрам напрямую. Помимо регистров, процессор содержит другое аппаратное обеспечение, необходимое для работы, но программисту эти компоненты доступны только посредством команд.

Цикл процессора

Работа процессора 8088 (равно как и всех других вычислительных машин) сводится к исполнению команд в определенной последовательности. Процесс исполнения отдельной команды делится на несколько этапов:

1. С помощью регистра PC производится выборка команды из кодового сегмента памяти.
2. К текущему значению счетчика команд прибавляется единица.
3. Выбранная команда декодируется.
4. Все необходимые для исполнения команды данные выбираются из памяти и (или) регистров процессора.
5. Команда исполняется.
6. Результаты исполнения команды сохраняются в памяти и (или) регистрах.
7. Начинается процесс исполнения следующей команды (переход к шагу 1).

Исполнение команды похоже на исполнение очень маленькой программы. Более того, в некоторых машинах для исполнения команд действительно предусмотрена небольшая программа (так называемая **микропрограмма**). Микропрограммы в подробностях рассматриваются в главе 4.

С точки зрения программиста, пишущего на ассемблере, в процессоре 8088 предусмотрено 14 регистров. Эти регистры выполняют функцию сверхоперативной памяти, с которой работают многочисленные команды; результаты, сохраняемые в них, меняются с очень большой частотой. Все эти регистры изображены на рис. В.2. Совершенно очевидно сходство между этим рисунком и окном трассера, приведенным на рис. В.1.

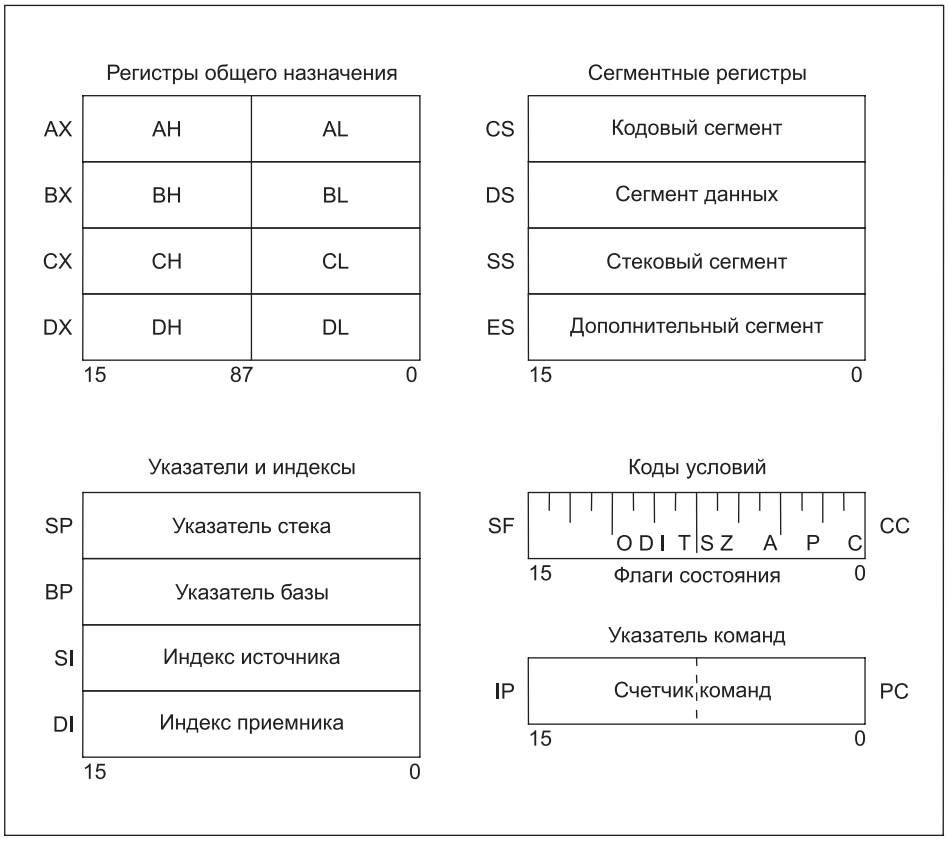


Рис. В.2. Регистры процессора 8088

Разрядность регистров процессора 8088 составляет 16 бит. Не существует ни одной пары полностью функционально идентичных регистров. В то же время некоторые из них близки по некоторым своим характеристикам, в связи с чем их подразделяют не несколько групп, что также отражено на рис. В.2. Эти группы мы сейчас и обсудим.

Регистры общего назначения

Регистры AX, BX, CX и DX входят в группу **регистров общего назначения**. Первый регистр в этой группе, AX, называется **регистром-сумматором**. Он применяется для накапливания результатов вычислений и часто выступает в роли приемника результатов исполнения различных команд. Хотя каждый регистр способен решать множество разнообразных задач, результаты исполнения некоторых команд (в частности, команд умножения) отправляются в регистр AX по умолчанию.

Второй регистр этой группы, BX, называется **базовым регистром**. По назначению он во многом аналогичен регистру AX, но есть и одно существенное отличие. В BX можно записать адрес памяти, а затем исполнить команду, операнд которой располагается по этому адресу памяти. Другими словами, BX может содержать указатель на область памяти, а AX — не может. Чтобы проиллюстрировать это утверждение, сравним две команды. Первая команда:

```
MOV AX,BX
```

Эта команда копирует содержимое BX в AX. Вторая команда:

```
MOV AX,(BX)
```

Эта команда копирует в AX содержимое слова памяти, адрес которого содержится в BX. В первом примере исходный операнд содержится в регистре BX; во втором примере фигурирует указатель на исходный операнд. В обоих примерах, как видите, для команды MOV заданы оба операнда: исходный (источник) и целевой (приемник). При этом целевой операнд указывается перед исходным.

Следующий регистр общего назначения, CX, называется **регистром-счетчиком**. Помимо прочего, он применяется для хранения значений счетчиков при исполнении циклов. Во время обработки команды LOOP значение в этом регистре автоматически уменьшается на единицу. Как правило, циклы завершаются в тот момент, когда значение в регистре CX достигает нуля.

Четвертым регистром в группе регистров общего назначения является **регистр данных** (DX). Совместно с регистром AX он задействуется при исполнении команд со словами двойной длины (32-разрядными). В этом случае в DX сохраняются старшие 16 бит, а в AX — младшие 16 бит. Здесь нужно оговориться, что обычно 32-разрядные целые числа называются **длинными**. Термин **двойные** чаще применяется по отношению к 64-разрядным значениям с плавающей точкой, но иногда так называют и 32-разрядные целочисленные значения. В нашем контексте путаницы не возникнет, так как числа с плавающей точкой мы обсуждать не собираемся.

Каждый регистр общего назначения можно рассматривать либо как один 16-разрядный регистр, либо как пару 8-разрядных регистров. Таким образом, в процессоре 8088 предусмотрено восемь 8-разрядных регистров, применяемых при исполнении команд с байтами и символами. Регистры, входящие во

все остальные группы, нельзя разделить на две 8-разрядные части. Одни команды задействуют целый регистр (например, AX), другие — только одну его часть (например, AL или AH). Согласно общему правилу, те команды, которые выполняют арифметические операции, обычно используют целые 16-разрядные регистры, а те, что имеют дело с символами, чаще всего довольствуются 8-разрядными регистрами. Следует иметь в виду, что AL и AH — не что иное, как имена двух половин регистра AX. При записи в AX нового 16-разрядного числа в AL и AH размещаются его нижняя и верхняя половины соответственно. Взаимодействие регистров AX, AH и AL можно проиллюстрировать следующей командой:

```
MOV AX,258
```

Она загружает в регистр AX десятичное значение 258. После завершения этой команды в байтовом регистре AH оказывается значение 1, а в байтовом регистре AL — значение 2. Пусть следом за этой командой следует другая:

```
ADDB AH,AL
```

В этом случае к значению байтового регистра AH прибавляется значение AL (то есть 2), и результирующее значение становится равным 3. В результате этой операции в регистр AX записывается новое значение — 770, эквивалентное значению 00000011 00000010 в двоичной системе счисления или значению 0x03 0x02 в шестнадцатеричной системе счисления. Как правило, два регистра по 8 байт взаимозаменяемы. Исключение составляет лишь команда MULB, при исполнении которой один из операндов всегда сохраняется в регистре AL, который вместе с AH является в этом случае приемником. При исполнении команды DIVB в паре регистров AH и AL сохраняется делимое. Нижний байт регистра-счетчика CL применяется для хранения числа циклов при исполнении команд обычного и циклического сдвига.

Во втором примере из раздела «Примеры» (этот пример относится к программе GenReg.s) представлены некоторые свойства регистров общего назначения.

Регистры-указатели

Во второй группе регистров содержатся **регистры-указатели** и **индексные регистры**. Наиболее важным регистром из этой группы считается **указатель стека** (SP). Стеки играют важную роль в большинстве языков программирования. Стек — это сегмент памяти, в котором хранятся те или иные данные, связанные с контекстом исполняемой программы. Как правило, при вызове процедуры часть стека резервируется для хранения ее локальных переменных, адреса, по которому нужно будет возвратиться по окончании процедуры, и ряда других данных управления. Часть стека, связанная с исполнением процедуры, называется **стековым кадром**. Когда ранее вызванная процедура вызывает другую процедуру, выделяется дополнительный стековый кадр, который, как правило, размещается непосредственно под первым. Соответственно, при вызове всех последующих процедур для каждой из них в нисходящем порядке выделяются новые стековые кадры. Не всегда, но в большинстве случаев стеки прирастают по нисходящей — от больших адресов к меньшим. Тем не менее вершиной стека называется наименьший по величине адрес.

Помимо локальных переменных, в стеках хранятся временные результаты исполнения. В процессоре 8088 есть команда PUSH, которая помещает 16-разрядное слово на вершину стека. Для этого она сначала уменьшает значение регистра SP на 2, а затем сохраняет свой операнд по адресу, на который этот регистр указывает после обновления. Команда POP удаляет 16-разрядные слова с вершины стека аналогичным образом — выбирает размещенное на этой вершине значение и прибавляет к содержимому регистра SP двойку. Регистр SP, указывающий на вершину стека, можно изменить командами PUSH, POP и CALL; при этом PUSH и CALL уменьшают его значение, а POP, наоборот, увеличивает.

Следующий регистр в этой группе называется **указателем базы** (BP). Обычно в него записывается некий адрес в стеке. В отличие от регистра SP, который всегда указывает на вершину стека, BP может указывать на любое место в стеке. Чаще всего регистр BP применяется для указания на начало стекового кадра текущей процедуры; тем самым упрощается задача поиска ее локальных переменных. Таким образом, BP часто указывает дно текущего стекового кадра (иными словами, на слово в стековом кадре с наименьшим числовым значением), а SP — на его вершину (на слово в стековом кадре с наибольшим числовым значением). Следовательно, границы текущего стекового кадра определяются значениями регистров-указателей BP и SP.

В этой группе есть два индексных регистра: SI (Source Index — **индекс источника**) и DI (Destination Index — **индекс приемника**). В сочетании с BP эти регистры часто употребляются для адресации данных в стеке, а в сочетании с BX — для вычисления адресов памяти. Более подробно о применении этих регистров мы поговорим в разделе, посвященном режимам адресации.

Один из важнейших регистров, составляющий самодостаточную группу, называется **указателем команд** – так Intel называет счетчик команд (PC). Команды обращаются к нему за адресами кодовых сегментов в памяти. Цикл исполнения команд процессором начинается с выборки команды, на которую указывает регистр PC. Перед выполнением последующих команд цикла к значению этого регистра прибавляется единица. Таким образом, счетчик команд всегда указывает на первую команду вслед за текущей.

Флаговый регистр, или **регистр кода условия**, по существу, представляет собой целый набор регистров, по одному биту каждый:

- Z — нулевой результат;
- S — отрицательный результат (знаковый бит);
- V — результат породил переполнение;
- C — результат породил перенос;
- A — служебный перенос (из бита 3);
- P — четность результата.

Другие биты в этом регистре регулируют различные аспекты работы процессора. В частности, бит I включает прерывания, а бит T — режим трассировки, применяемый для отладки программ. Наконец, бит D регулирует направление строковых операций. Не все 16 бит флагового регистра задействованы; неиспользуемые биты имеют фиксированное нулевое значение.

К **группе сегментных регистров** причисляется четыре регистра. Как вы помните, стек, данные и коды команд хранятся в разных областях основной памяти. Сегментные регистры объединяют эти области памяти, называемые **сегментами**. К сегментным регистрам относятся регистр кодового сегмента (CS), регистр сегмента данных (DS), регистр стекового сегмента (SS) и регистр дополнительного сегмента (ES). Большую часть времени их значения остаются неизменными. На самом деле, сегмент данных и стековый сегмент относятся к одной и той же области памяти, но данные хранятся на дне этого общего сегмента, а стек — на вершине. Более подробно о сегментах мы поговорим в подразделе «Организация памяти и сегменты» раздела «Память и адресация».

Память и адресация

Память процессора 8088 организована достаточно необычно, что объясняется сочетанием 1-мегабайтной памяти и 16-разрядных регистров. Дело в том, что в памяти емкостью 1 Мбайт для представления адреса требуется 20 бит. Следовательно, сохранить указатель на элемент памяти в одном 16-разрядном регистре невозможно. В целях решения проблемы память поделена на сегменты по 64 Кбайт каждый, и адреса в рамках этих сегментов умещаются в 16 бит. Далее мы рассмотрим архитектуру памяти 8088 более подробно.

Организация памяти и сегменты

Память процессора 8088, состоящая из массива адресуемых 8-разрядных байтов, применяется для хранения команд, данных и стека. Для разделения областей памяти, применяемых для разных целей, в процессоре 8088 вводится понятие **сегментов**, которые представляют собой отделенные друг от друга блоки памяти. Такой сегмент в 8088 состоит из 65 536 последовательных байтов. Всего сегментов четыре: кодовый, данных, стековый и дополнительный.

В кодовом сегменте содержатся команды, из которых состоят программы. Содержимое регистра PC всегда интерпретируется как адрес кодового сегмента в памяти. Нулевое значение PC указывает не на абсолютный нулевой адрес в памяти, а на нижний адрес в данном кодовом сегменте. В сегменте данных сохраняются инициализированные и неинициализированные данные программы. Если в регистре BX содержится указатель, он обязательно указывает на сегмент данных. В стековом сегменте содержатся локальные переменные и промежуточные результаты, помещенные в стек. Адреса, указанные в регистрах SP и BP, всегда относятся к стековому сегменту. Дополнительный сегмент — это вспомогательный сегментный регистр, который по необходимости можно разместить в произвольном месте в памяти.

Каждому из этих сегментов соответствует один из 16-разрядных сегментных регистров CS, DS, SS и ES. Начальным адресом сегмента является 20-разрядное целое число без знака, формируемое путем сдвига сегментного регистра на 4 бита влево и размещения в четырех освободившихся позициях справа четырех нулей. Следовательно, сегментные регистры в рамках 20-разрядного адресного пространства всегда выражаются числами, кратными 16-ти. Сегментный регистр

указывает на базу сегмента. Адреса сегментов формируются путем преобразования 16-разрядного значения в сегментном регистре в фактический 20-разрядный адрес — для этого в конец значения прибавляется четыре нулевых бита и выполняется смещение. В итоге абсолютный адрес в памяти устанавливается путем умножения значения в сегментном регистре на 16 и прибавления смещения. К примеру, если значение DS равняется 7, а BX — 12, значит, BX указывает на адрес $7 \times 16 + 12 = 124$. Иными словами, на основании значения 7 в регистре DS можно получить 20-разрядный двоичный адрес 00000000000001110000. Добавив к исходному значению сегмента 16-разрядное смещение 0000000000001100 (десятичное 12), получаем 20-разрядный адрес 00000000000001111100 (десятичное 124).

При *каждом* обращении к памяти один из сегментных регистров применяется для формирования фактического адреса в памяти. Если та или иная команда содержит непосредственный адрес без указания на регистр, считается, что этот адрес находится в сегменте данных, и для определения базы этого сегмента используется регистр DS. Физический адрес определяется путем сложения результата предыдущей операции с адресом, указанным в команде. Физический адрес в памяти кода следующей команды устанавливается путем смещения содержимого регистра CS на четыре двоичных разряда и сложения со значением счетчика команд. Иными словами, сначала на основе значения 16-разрядного регистра CS вычисляется фактический 20-разрядный адрес, а затем к нему прибавляется новое 16-разрядное значение PC; в результате получается 20-разрядный абсолютный адрес в памяти.

Стековый сегмент состоит из 2-байтовых слов, а это значит, что в указателе стека (SP) может храниться только четное число. Стек заполняется в порядке от больших адресов к меньшим. Таким образом, команда PUSH уменьшает значение указателя стека на 2, а затем сохраняет операнд по адресу в памяти, вычисленному на основании значений SS и SP. Команда POP извлекает полученное после исполнения операции PUSH значение и увеличивает значение SP на 2. Те адреса в стековом сегменте, которые расположены ниже предела, определяемого регистром SP, считаются свободными. Следовательно, очистка стека осуществляется за счет одного лишь приращения SP. На практике значения регистров DS и SS всегда совпадают, так что для обращения к переменной в общем сегменте данных и стека достаточно 16-разрядного указателя. Если бы значения регистров DS и SS различались, к каждому указателю потребовалось бы прибавлять 17-й бит — только так можно было бы различить указатели на сегмент данных и на стековый сегмент. По большому счету, создание разработчиками микросхемы отдельного стекового сегмента вряд ли можно признать оправданным.

Если адреса в четырех сегментных регистрах значительно отстоят друг от друга, эти сегменты разделяются, однако в условиях ограниченного объема доступной памяти разделять их не обязательно. Объем программного кода после компиляции неизвестен. Поэтому эффективнее всего размещать начальную границу сегмента данных и стекового сегмента через первое кратное 16-ти значение после последней команды. В этом решении подразумевается, что кодовый сегмент и сегмент данных ни при каких обстоятельствах не будут использовать один и тот же физический адрес.

Адресация

Для исполнения абсолютного большинства команд требуются данные, которые извлекаются либо из памяти, либо из регистров. Для именования этих данных в 8088 предусмотрено несколько режимов адресации. Многие команды содержат по два операнда, которые в этом случае обычно называются **исходным** (источник) и **целевым** (приемник). Возьмем для примера команды копирования и сложения:

```
MOV AX,BX
ADD CX,20
```

Первые операнды в этих командах являются приемниками, а вторые — источниками. (Их очередность не принципиальна; она с тем же успехом могла бы быть обратной.) Очевидно, что в таком случае приемник является **левым значением**, то есть в этом месте должно располагаться перезаписываемое значение. Следовательно, целевые операнды в отличие от исходных не могут быть константами.

В первоначальном варианте архитектуры 8088 один из операндов в случае, если в команде их два, обязательно должен был быть регистром. Предполагалось, что это требование позволит различать **команды обработки байтов** и **слов**, исходя из того, является ли адресуемый регистр регистром для байтов или для слов. В первой версии процессора это правило соблюдалось с неимоверной строгостью. Невозможно было даже ввести в стек константу, поскольку в таком случае ни в одном из двух операндов команды не было бы ни одного регистра. В последующих версиях ограничения были смягчены, но сам принцип оказал определенное влияние на конструктивное решение процессора. В некоторых случаях один из двух операндов не упоминается. Например, в команде `MULB` роль приемника может исполнять только регистр AX.

Существуют также команды с одним операндом. В эту категорию входят, помимо прочего, команды приращения, сдвига и сброса. В этих командах требование о применении регистра в качестве операнда не действует, и различать команды обработки байтов и слов можно только по кодам операций (то есть типам команд).

Процессор 8088 поддерживает четыре базовых типа данных: **байт**, **слово** (2 байта), **длинное слово** (4 байта) и **двоично-десятичное число**, в котором в слово упаковываются две десятичные цифры. Последний тип не поддерживается интерпретатором.

Адрес в памяти всегда соответствует байту, но в случае с обычным или длинным словом также присутствует неявное указание на ячейки памяти, находящиеся непосредственно над указанным байтом. К примеру, слово, расположенное по адресу 20, занимает ячейки памяти 20 и 21. Длинное слово по адресу 24 занимает ячейки 24–27. Процессор 8088 является системой **с обратным порядком следования байтов** (little endian); это означает, что часть слова меньшего порядка хранится по младшему адресу. В стековом сегменте слова должны размещаться по четным адресам. Комбинация AX DX, где в AX содержится слово меньшего порядка, является единственно возможным вариантом размещения длинных слов в регистрах процессора 8088.

В табл. В.1 представлены все предусмотренные в 8088 режимы адресации. Далее мы вкратце их рассмотрим. В верхних строках таблицы перечислены регистры. Они могут быть задействованы в качестве источников или приемников практически во всех командах. Имеются 8 регистров для слов и столько же для байтов.

Таблица В.1. Режимы адресации операндов (символ # означает численное значение или метку)

Режим адресации	Операнд	Примеры
Регистровая адресация		
По регистру для байтов	Регистр для байтов	AH, AL, BH, BL, CH, CL, DH, DL
По регистру для слов	Регистр для слов	AX, BX, CX, DX, SP, BP, SI, DI
Адресация сегментов данных		
Непосредственная адресация	Адрес после кода операции	(#)
Косвенная регистровая адресация	Адрес в регистре	(SI), (DI), (BX)
Регистровая адресация со смещением	Адрес формируется по значению регистра со смещением	#(SI), #(DI), #(BX)
Индексная регистровая адресация	Адрес формируется путем сложения BX с SI/DI	(BX)(SI), (BX)(DI)
Индексная регистровая адресация со смещением	BX плюс SI DI плюс смещение	#(BX)(SI), #(BX)(DI)
Адресация стекового сегмента		
Косвенная адресация по указателю базы	Адрес в регистре	(BP)
Смещение указателя базы	Адрес формируется по значению BP и смещения	#(BP)
Адресация по указателю базы с индексом	Адрес формируется путем сложения BP и SI/DI	(BP)(SI), (BP)(DI)
Смещение индекса указателя базы	BP плюс SI/DI плюс смещение	#(BP)(SI), #(BP)(DI)
Непосредственная адресация данных		
По непосредственному байту/слову	Информационная часть команды	#
Неявная адресация		
Команда введения в стек или выталкивания из стека	Косвенный адрес (SP)	PUSH, POP, PUSHF, POPF

Режим адресации	Операнд	Примеры
Флаги загрузки или сохранения	Регистр с флагами состояния	LAHF, STC, CLC, CMC
Трансляция XLAT	AL, BX	XLAT
Повторяющиеся строковые команды	(SI), (DI), (CX)	MOVS, CMPS, SCAS
Входные и выходные команды	AX, AL	IN #, OUT #
Преобразование байта, слова	AL, AX, DX	CBW, CWD

В строках под заголовком «Адресация сегментов данных» перечислены режимы адресации, актуальные для сегментов данных. Адреса этого типа всегда заключаются в пару круглых скобок — именно они позволяют отличить адрес в их составе от значения иного типа. Простейшим режимом адресации из этой категории является **непосредственная адресация**, при которой адрес данных операнда включается в команду. Пример:

```
ADD CX,(20)
```

В данном случае к регистру CX прибавляется содержимое слова памяти, расположенного по адресам 20 и 21. В языке ассемблера ячейки обычно выражаются метками, а не численными значениями, а преобразование выполняется уже в период ассемблирования. Даже в командах CALL и JMP целевой операнд можно сохранить в ячейке памяти, на которую указывает метка. Скобки вокруг меток совершенно необходимы (по крайней мере, для того ассемблера, о котором мы ведем речь), поскольку следующая команда тоже имеет право на существование:

```
ADD CX,20
```

Однако эта команда выражает совершенно другую операцию, а именно — прибавление к значению регистра CX константы 20, а не содержимого слова памяти, находящегося по адресу 20. Символ # в табл. В.1 может обозначать численную константу, метку или константное выражение с меткой.

При **косвенной регистровой адресации** адрес операнда хранится в одном из трех регистров: BX, SI или DI. Во всех этих случаях он оказывается в сегменте данных. Кроме того, можно разместить константу перед регистром, и в этом случае адрес устанавливается путем сложения регистра с константой. Этот тип адресации, называемый **регистровой адресацией со смещением**, удобен при обработке массивов. Так, если регистр SI содержит значение 5, то пятый символ строки, определяемой меткой FORMAT, можно загрузить в регистр AL посредством оператора:

```
MOVB AL,FORMAT(SI)
```

При этом просмотр всей строки осуществляется на каждом этапе путем положительного или отрицательного приращения значения регистра. При использовании операндов-слов значение регистра каждый раз изменяется в ту или иную сторону на 2.

Кроме того, можно поместить базу (то есть нижний числовой адрес) массива в регистр BX, сохранив регистры SI или DI для отсчета. Такая схема называется **индексной регистровой адресацией**. Пример:

```
PUSH (BX)(DI)
```

Этот код извлекает содержимое ячейки из сегмента данных с адресом, который определяется суммой регистров BX и DI. Результирующее значение затем помещается в стек. Сочетание последних двух типов адресации дает **индексную регистровую адресацию со смещением**:

```
NOT 20(BX)(DI)
```

Эта команда обращается к слову памяти по адресам BX + DI + 20 и BX + DI + 21.

Все способы косвенной адресации, применимые к сегменту данных, актуальны и для стекового сегмента; в последнем случае базовый регистр BX заменяется указателем базы BP. Таким образом, `(BP)` — это только режим косвенной адресации стека регистров, однако существуют и более сложные режимы, в частности косвенная адресация по указателю базы с индексом и смещением: `-1(BP)(SI)`. Эти режимы полезны при обращении к локальным переменным и параметрам функций, хранящихся по адресам стека в подпрограммах. Соответствующая схема описывается в подразделе «Вызовы подпрограмм».

Все адреса, соответствующее перечисленным режимам адресации, в операциях могут выступать в качестве как источников, так и приемников. Взятые вместе, эти две категории называются **действительными адресами**. Режимы адресации двух оставшихся категорий не предполагают наличие приемников, поэтому соответствующие адреса не входят в число действительных. Они применяются только в качестве источников.

Режим адресации, при котором операндом является константный байт или значение слова, называется **непосредственной адресацией**. Например:

```
CMP AX,50
```

Эта команда сравнивает значение в регистре AX с константой 50 и в зависимости от результата устанавливает биты во флаговом регистре.

Наконец, некоторые команды используют **неявную адресацию**. В таких командах операнд или операнды лишь подразумеваются. К примеру:

```
PUSH AX
```

Эта команда вводит содержимое регистра AX в стек; для этого она уменьшает значение SP на единицу, а затем копирует содержимое AX в ячейку, на которую теперь указывает SP. При этом регистр SP в команде не упоминается. Аналогичным образом, регистр флагов состояния не упоминается в командах управления флагами. Неявные операнды применяются и в других командах.

В процессоре 8088 предусмотрены специальные команды для перемещения (`MOVS`), сравнения (`CMPS`) и просмотра (`SCAS`) строк. После выполнения этих строковых команд содержимое индексных регистров SI и DI автоматически обновляется. Этот механизм называется, в зависимости от направления, **автоинкрементным** или **автодекрементным** режимом. Направление приращения (положительное или отрицательное) регистров SI и DI определяется **флагом направления** в регистре флагов состояния. Флаг направления с нулевым значением определяет положительное приращение, а флаг с единичным значением — отрицательное. Величина

приращения составляет 1 в командах для байтов и 2 в командах для слов. В некотором смысле указатель стека также имеет отношение к автоинкрементному и автодекрементному режимам: его значение уменьшается на 2 в начале операции PUSH и увеличивается на 2 по окончании операции POP.

Набор команд 8088

Любой компьютер характеризуется, в первую очередь, набором команд, которые он способен исполнять. Чтобы понять, как работает компьютер, необходимо основательно изучить его набор команд. В этом разделе мы обсудим наиболее важные команды 8088. Некоторые из этих команд представлены в табл. B.2, где они подразделяются на 10 групп.

Таблица B.2. Некоторые из наиболее важных команд процессора 8088

Мнемоника	Описание	Операнды	Флаги состояния			
			O	S	Z	C
MOV(B)	Перемещение слова, байта	r←e, e←r, e←#	—	—	—	—
XCHG(B)	Обмен словами	r←→e	—	—	—	—
LEA	Загрузка действительного адреса	r←#e	—	—	—	—
PUSH	Введение в стек	e, #	—	—	—	—
POP	Выталкивание из стека	e	—	—	—	—
PUSHF	Введение в стек флагов	—	—	—	—	—
POPF	Выталкивание из стека флагов	—	—	—	—	—
XLAT	Трансляция AL	—	—	—	—	—
ADD(B)	Сложение слова	r←e, e←r, e←#	*	*	*	*
ADC(B)	Сложение слова с переносом	r←e, e←r, e←#	*	*	*	*
SUB(B)	Вычитание слова	r←e, e←r, e←#	*	*	*	*
SBB(B)	Вычитание слова с отрицательным переносом	r←e, e←r, e←#	*	*	*	*
IMUL(B)	Умножение с учетом знака	e	*	U	U	*
MUL(B)	Умножение без учета знака	e	*	U	U	*
IDIV(B)	Деление с учетом знака	e	U	U	U	U
DIV(B)	Деление без учета знака	e	U	U	U	U

продолжение ⇩

Таблица В.2 *(продолжение)*

Мнемоника	Описание	Операнды	Флаги состояния			
			O	S	Z	C
CBW	Дополнение байта до слова с учетом знака	—	—	—	—	—
CWD	Дополнение слова до двойного слова с учетом знака	—	—	—	—	
NEG(B)	Отрицание двоичного числа	e	*	*	*	*
NOT(B)	Логическое отрицание	e	—	—	—	—
INC(B)	Положительное приращение по целевому адресу	e	*	*	*	—
DEC(B)	Отрицательное приращение по целевому адресу	e	*	*	*	—
AND(B)	Логическое И	e←r, r←e, e←#	0	*	*	0
OR(B)	Логическое ИЛИ	e←r, r←e, e←#	0	*	*	0
XOR(B)	Логическое исключающее ИЛИ	e←r, r←e, e←#	0	*	*	0
SHR(B)	Логический сдвиг вправо	e←1, e←CL	*	*	*	*
SAR(B)	Арифметический сдвиг вправо	e←1, e←CL	*	*	*	*
SAL(B) (=SHL(B))	Сдвиг влево	e←1, e←CL	*	*	*	*
ROL(B)	Циклический сдвиг влево	e←1, e←CL	*	—	—	*
ROR(B)	Циклический сдвиг вправо	e←1, e←CL	*	—	—	*
RCL(B)	Циклический сдвиг влево с переносом	e←1, e←CL	*	—	—	*
RCR(B)	Циклический сдвиг вправо с переносом	e←1, e←CL	*	—	—	*
TEST(B)	Проверка операндов	e←→r, e←→#	0	*	*	0
CMP(B)	Сравнение операндов	e←→r, e←→#	*	*	*	*
STD	Установка флага направления (↓)	—	—	—	—	—
CLD	Сброс флага направления (↑)	—	—	—	—	—
STC	Установка флага переноса	—	—	—	—	1
CLC	Сброс флага переноса	—	—	—	—	0

Мнемоника	Описание	Операнды	Флаги состояния			
			O	S	Z	C
CMC	Обратный перенос	—	—	—	—	*
LOOP	Обратный переход, если DEC(CX) □ 0	Метка	—	—	—	—
LOOPZ LOOPE	Обратный переход, если Z = 1 и DEC(CX) □ 0	Метка	—	—	—	—
LOOPNZ LOOPNE	Обратный переход, если Z = 0 и DEC(CX) □ 0	Метка	—	—	—	—
REP REPZ REPNZ	Повтор строковой команды	Строковая команда	—	—	—	—
MOVS(B)	Перемещение строки слов	—	—	—	—	—
LODS(B)	Загрузка строки слов	—	—	—	—	—
STOS(B)	Сохранение строки слов	—	—	—	—	—
SCAS(B)	Просмотр строки слов	—	*	*	*	*
CMPS(B)	Сравнение строк слов	—	*	*	*	*
JCC	Условный переход	Метка	—	—	—	—
JMP	Переход к метке	e, метка	—	—	—	—
CALL	Переход к подпрограмме	e, метка	—	—	—	—
RET	Возврат из подпрограммы	—, #	—	—	—	—
SYS	Вызов системного исключения	—	—	—	—	—

Перемещение, копирование и арифметические команды

В первую группу входят команды копирования и перемещения. Крайне важна команда `MOV`, в которой явно указываются исходный и целевой адреса. Если в качестве исходного адреса выступает регистр, целевой адрес может быть действительным. В табл. В.2 регистровые операнды обозначаются символом *r*, а действительные адреса — символом *e*. Соответственно, сочетание, о котором идет речь, выглядит как *e*←*r*. Именно это обозначение идет первым в ячейке операндов команды `MOV`. Поскольку, согласно синтаксису команд, целевой адрес должен быть первым операндом, а исходный адрес — вторым, при обозначении операндов используется символ направленной влево стрелки (←). Таким образом, запись *e*←*r* означает, что содержимое регистра копируется по действительному адресу.

Кроме того, в команде `MOV` действительный адрес может быть исходным, а регистр — целевым. Такая ситуация обозначается как *r*←*e* (это вторая запись в ячейке операндов упомянутой команды). Третий вариант — по исходному адресу находятся сами данные, а целевой адрес является действительным, что выражается как *e*←#. Непосредственно данные в таблице обозначаются символом

решетки (#). Символ «B» в скобках в конце мнемонического кода призван показать, что существуют команды перемещения слова (MOV) и байта (MOVB). Таким образом, данная строка в таблице реально описывает шесть разных команд.

Ни один из флагов в регистре кода условий не меняется в зависимости от исполнения команды перемещения, и поэтому в последних четырех столбцах показан прочерк (—). Обратите внимание, что команды перемещения на самом деле не перемещают данные. Они создают копии, не изменяя исходных данных, и именно в этом состоит отличие от традиционной операции перемещения.

Вторая команда, представленная в этой таблице, — XCHG. Она меняет местами содержимое регистра и действительного адреса. Для обозначения операции обмена в таблице применяется символ двунаправленной стрелки (←→). Поскольку в данном случае существуют варианты операции с байтом и со словом, в поле операндов команды XCHG содержится обозначение *r*←→*e*. Следующей в таблице указывается команда загрузки действительного адреса (LEA). Она определяет численное значение действительного адреса и сохраняет его в регистре.

Далее следует команда PUSH, которая вводит свой операнд в стек. Ее явный операнд может быть либо константой (что обозначается символом «#» в столбце операндов), либо действительным адресом (символ «e» в столбце операндов). Кроме того, имеет место неявный операнд SP, который не указывается в синтаксисе команды. Команда уменьшает значение SP на 2, а затем сохраняет операнд по адресу, на который SP указывает после обновления.

Следующая команда — POP — удаляет операнд из стека и размещает его по действительному адресу. Команды PUSHF и POPF, выполняющие введение в стек и выталкивание из стека регистра флагов состояния, также предусматривают наличие неявных операндов. Аналогично обстоит дело и с командой XLAT, которая загружает байтовый регистр AL с адреса, формируемого сложением AL и BX. Эта команда позволяет проводить быстрый поиск в таблицах размером 256 байт.

Определенные в официальной спецификации 8088 команды IN и OUT не реализованы в интерпретаторе (и по этой причине не указаны в табл. B.2). По факту, это команды перемещения данных в устройство ввода-вывода и из него. Неявным адресом в них всегда является регистр AX, а вторым операндом команд выступает номер порта регистра целевого устройства.

Во второй группе табл. B.2 представлены команды сложения и вычитания. Для всех них характерно то же сочетание трех операндов, что и для MOV: от действительного адреса к регистру (*r*←*e*), от регистра к действительному адресу (*e*←*r*) и от константы к действительному адресу (*e*←#). Во всех четырех командах в зависимости от результата исполнения могут быть установлены флаг переполнения (*O*), знаковый флаг (*S*), нулевой флаг (*Z*) и флаг переноса (*C*). Флаг *O* устанавливается в том случае, если результат невозможно с достаточной степенью точности выразить разрешенным числом битов, и сбрасывается, если такая возможность существует. Скажем, при прибавлении максимально допустимого 16-разрядного числа 0x7fff (32 767 в десятеричной системе) к самому себе результат нельзя выразить в виде 16-разрядного числа со знаком, и в этом случае для указания на ошибку устанавливается флаг *O*. Аналогичным образом устроен механизм установки других флагов состояния. Если команда способна влиять на флаг состояния, в соответствующем столбце это обстоятельство обозначается звездочкой (*). В командах ADC и SBB флаг переноса в начале операции исполняет

роль дополнительной единицы (или нуля), выражающей положительный или отрицательный перенос после предыдущей операции. Эта возможность особенно полезна для представления 32-разрядных и более длинных целочисленных значений в нескольких словах. Помимо вышеперечисленного, предусмотрены операции сложения и вычитания байтов.

В следующем блоке таблицы содержатся команды умножения и деления. Для работы с целочисленными операндами со знаком нужны команды IMUL и IDIV; для обработки значений без знака достаточно команд MUL и DIV. В байтовых вариантах этих команд в качестве целевого адреса неявно принимается комбинация регистров AH : AL. В командах обработки слов неявным целевым адресом выступает комбинация регистров AX : DX. Даже если результатом умножения оказывается одно слово или один байт, значение регистра DX или AX обновляется. Недопустимых операций умножения не бывает, поскольку нехватка битов по целевому адресу исключена. Биты переполнения и переноса устанавливаются тогда, когда произведение невозможно представить одним словом или одним байтом. Нулевой и отрицательный флаги оказываются после умножения неопределенными.

При делении в качестве целевых адресов используются те же комбинации регистров: DX : AX или AH : AL. Частное записывается в AX или AL, а остаток — в DX или AH. Все четыре флага — переноса, переполнения, нуля и отрицательности — после выполнения операции деления переводятся в неопределенное состояние. При нулевом делителе, а также в том случае, когда частное не помещается в регистр, инициируется **исключение**, и если процедура перехвата исключений в программе не предусмотрена, программа останавливается. Больше того, затруднена программная обработка знаков «минус» перед или после операции деления, поскольку согласно спецификации 8088, знак остатка должен быть всегда равен знаку делимого, в то время как в математике допускается только неотрицательный остаток.

Команды обработки двоично-десятичных чисел, в том числе команды коррекции перед сложением ASCII-символов (AAA) и десятичных значений (DAA), не реализованы в интерпретаторе и не представлены в табл. В.2.

Логические операции, побитовые операции и операции сдвига

В следующей группе содержатся команды знакового расширения, отрицания, логического отрицания, положительного и отрицательного приращений. Операции знакового расширения, не располагающие явными операндами, работают с комбинациями регистров DX : AX или AH : AL. Применительно ко всем остальным операциями из этой группы операнды (единичные) могут располагаться по любым действительным адресам. Исполнение операций NEG, INC и DEC определяет состояние флагов стандартным способом, однако при отрицательном и положительном приращении флаг переноса не изменяется. Некоторые специалисты считают это странное последнее обстоятельство проектной ошибкой.

Следующая группа команд содержит логические команды с двумя операндами. Все они исполняются стандартным способом. В группе команд обычного и циклического сдвига у всех операций в роли целевых адресов выступают действительные

адреса; исходный адрес представлен в виде байтового регистра CL или единицы. Исполнение операций сдвига оказывает влияние на все четыре флага; при циклических сдвигах могут изменяться только флаги переноса и переполнения. После обычного или циклического сдвига переносимый разряд может оказаться либо старшим, либо младшим — в зависимости от направления обычного или циклического сдвига. При исполнении циклического сдвига с переносом командами RCR, RCL, RCRB и RCLB сочетание переноса и операнда, расположенного по действительному адресу, образует 17- или 9-разрядную комбинацию регистров кругового сдвига, которая упрощает обычные и циклические сдвиги с участием нескольких слов.

Команды, входящие в следующую группу, применяются для управления флаговыми разрядами. В основном, они нужны для подготовки к условным переходам. Двунаправленная стрелка (←→) в данном случае обозначает проведение операций сравнения и проверки с двумя неизменяемыми операндами. При исполнении команды TEST с операндами проводится операция логического И, по результатам которой устанавливаются или сбрасываются нулевой и знаковый флаги. Вычисленное значение при этом не сохраняется, а операнды остаются без изменений. Команда CMP сводится к вычислению разности операндов, в результате устанавливаются или сбрасываются все четыре флага. Флаг направления, регламентирующий отрицательное и положительное приращение значений регистров SI и DI при исполнении строковых команд, можно установить или сбрасывать командами STD и CLD соответственно.

В процессоре 8088 также предусмотрены **флаги четности** и **служебного переноса**. Флаг четности, как следует из его названия, показывает на четность или нечетность результата. Флаг служебного переноса позволяет проверить, не произошел ли перенос в нижнем (4-разрядном) полубайте целевого адрес. Кроме того, существуют команды LAHF и SAHF, которые копируют нижний байт флагового регистра в регистр AH, и наоборот. Флаг переполнения находится в старшем байте регистра кода условия и во время исполнения вышеупомянутых команд не копируется. Команды и флаги, о которых идет речь в этом абзаце, введены, в первую очередь, для обратной совместимости с процессорами 8080 и 8085.

Операции организации циклов и повторяющиеся строковые операции

В следующей группе содержатся команды организации циклов. Команда LOOP проводит отрицательное приращение регистра CX и в случае положительного результата выполняет обратный переход к указанной метке. Команды LOOPZ, LOOPE, LOOPNZ и LOOPNE, помимо прочего, проверяют значение нулевого флага на предмет необходимости прекращения цикла до достижения регистром CX нулевого значения.

Целевой адрес во всех командах категории LOOP должен отстоять от текущего положения счетчика команд не более чем на 128 байт; это связано с тем, что в командах предусмотрено 8-разрядное смещение со знаком. Количество *команд* (в отличие от байтов), через которые можно совершить переход, точно не определяется в силу различия между этими командами по длине. Как правило, первый байт задает тип команды, и во многих случаях содержание кодового сегмента команд им ограничивается. Второй байт часто определяет применяемые коман-

дой регистры и регистровый режим адресации, и если в команде используется смещение или содержатся непосредственные данные, ее длина увеличивается до четырех-шести байтов. Средняя длина команды составляет 2,5 байта, поэтому величина обратного перехода в циклах ограничивается примерно 50 командами.

Существует ряд специальных механизмов организации циклов строковых команд: REP, REPZ и REPNZ. Пять строковых команд, указанных в следующей группе табл. В.2, используют неявные адреса и автоинкрементный/автодекрементный режимы работы с индексными регистрами. Во всех этих командах регистр SI указывает на **сегмент данных**, а регистр DI — на **дополнительный сегмент**, основанный на содержании регистра ES. Как и REP, команда MOVSB позволяет единовременно перемещать целые строки. Длина строки определяется значением регистра CX. Так как команда MOVSB не влияет на состояние флагов, средствами REPNZ провести проверку на наличие байта ASCII-нуля при копировании не представляется возможным; чтобы исправить эту ситуацию, нужно сначала запустить команду REPNZ SCASB, позволяющую разместить в регистре CX осмысленное значение, а затем исполнить REP MOVSB. Пример практического применения этого механизма приводится в листинге кода копирования строки в разделе «Примеры». При работе со всеми упомянутыми командами следует уделять особое внимание сегментному регистру ES (если его значение не совпадает со значением регистра DS). В интерпретаторе используется малая модель памяти, так что ES = DS = SS.

Команды перехода и вызова

В последней группе табл. В.2 представлены команды условного и безусловного переходов, вызова подпрограмм и возврата. Простейшей из них является команда JMP. Метка в ней может указывать на целевой адрес или на содержимое любого действительного адреса. Следует учитывать различие между **ближним** и **дальним переходами**. Объектом ближнего перехода является текущий кодовый сегмент, неизменный в период исполнения операции. В процессе дальнего перехода значение регистра CS, напротив, меняется. В случае непосредственной адресации с применением метки новое значение регистра кодового сегмента устанавливается при вызове после метки; в случае с действительным адресом из памяти выбирается длинное слово, в результате младшее слово оказывается соответствующим целевой метке, а старшее слово — новому значению регистру кодового сегмента.

Ничего удивительного в таком различии нет. Чтобы выполнить переход к произвольному адресу в 20-разрядном адресном пространстве, необходим механизм определения адреса длиной более 16 бит. Этот механизм существует и заключается он в присвоении регистрам CS и PC новых значений.

Условные переходы

В процессоре 8088 предусмотрены 15 разновидностей условных переходов, причем у некоторых из них сразу несколько имен (например, команды JUMP GREATER OR EQUAL и JUMP NOT LESS THAN эквивалентны). Все они перечислены в табл. В.3. Максимальное расстояние перехода составляет 128 байт от текущей команды. Если объект перехода находится вне допустимого диапазона, приходится реализовывать составного перехода (jump over jump). В этом случае для перехода через

следующую команду применяется второй переход с противоположным условием. Если в следующей команде определен безусловный переход по целевому адресу, то сочетание этих двух команд приводит лишь к удлинению перехода указанного типа. К примеру:

```
JB FARLABEL
```

Эту команду следует заменить следующей конструкцией:

```
    JNA 1f
    JMP FARLABEL
1:
```

Иными словами, если исполнение команды `JUMP BELOW` невозможно, составляется конструкция, состоящая из команды `JUMP NOT ABOVE` с близлежащей меткой `1` в качестве объекта и безусловного перехода к `FARLABEL`. Результаты в обоих случаях одинаковы, различие заключается лишь в больших или меньших временных и пространственных затратах. Если объект перехода находится слишком далеко, ассемблер автоматически организует составной переход. Правильно провести подобные вычисления довольно сложно. Предположим, что расстояние до объекта близко к максимально допустимому, и при этом некоторые промежуточные команды содержат условные переходы. Внешний переход может быть проведен только после определения расстояний внутренних переходов. В такой ситуации ассемблер предпринимает определенные меры предосторожности. К примеру, он может сформировать составной переход без строгой на то необходимости. Непосредственный условный переход формируется ассемблером только в том случае, если объект перехода совершенно точно находится на досягаемом расстоянии.

Таблица В.3. Условные переходы

Команда	Описание	Условие перехода
JNA, JBE	Ниже или равно	CF = 1 или ZF = 1
JNB, JAE, JNC	Не ниже	CF = 0
JE, JZ	Нуль, равно	ZF = 1
JNLE, JG	Больше чем	SF = OF и ZF = 0
JGE, JNL	Больше или равно	SF = OF
JO	Переполнение	OF = 1
JS	Отрицательный знак	SF = 1
JCXZ	Значение CX равно нулю	CX = 0
JB, JNAE, JC	Ниже	CF = 1
JNBE, JA	Выше	CF = 0 и ZF = 0
JNE, JNZ	Не равно нулю, не равно	ZF = 0
JL, JNGE	Меньше чем	SF □ OF
JLE, JNG	Меньше или равно	SF □ OF или ZF = 1
JNO	Без переполнения	OF = 0
JNS	Неотрицательно	SF = 0

В большинстве своем условные переходы зависят от флагов состояния и предваряются командами сравнения или проверки. Команда CMP вычитает исходный операнд из целевого операнда, устанавливает коды состояния, после чего сбрасывает результат. Ни один из операндов при этом не меняется. При нулевом результате или установленном знаковом бите (обозначающем отрицательный результат) устанавливается соответствующий флаговый бит. Если результат не представляется возможным выразить допустимым числом битов, устанавливается флаг переполнения. Если старший бит сопровождается переносом, устанавливается флаг переноса. При условных переходах все эти биты можно проверить.

Для обработки операндов со знаками используются команды GREATER THAN и LESS THAN. Операнды без знаков обрабатываются командами ABOVE и BELOW.

Вызовы подпрограмм

В процессоре 8088 есть команда, позволяющая вызывать процедуры, которые в языке ассемблера обычно называются **подпрограммами** (subroutines). По аналогии с командами перехода здесь существуют команды **ближнего и дальнего вызова**. В интерпретаторе реализован только ближний вызов. Объект вызова либо является меткой, либо располагается по действительному адресу. Параметры, необходимые для вызова подпрограмм, первоначально размещаются в стеке в обратном порядке (рис. В.3). Применительно к языку ассемблера параметры обычно называются **аргументами**, хотя принципиального различия между этими терминами нет. После размещения аргументов в стеке исполняется команда CALL. В первую очередь, она вводит в стек текущее значение счетчика команд, сохраняя, таким образом, адрес возврата. Адресом возврата называется адрес, по которому возобновляется исполнение вызывающей программы после возврата из подпрограммы.

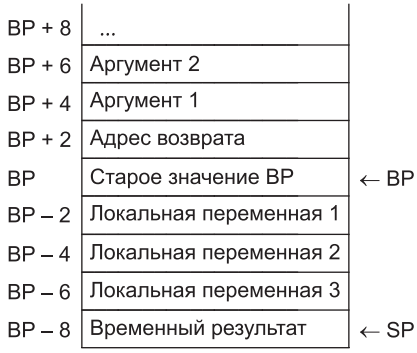


Рис. В.3. Пример стека

Далее из метки или с действительного адреса загружается новый счетчик команд. При дальнем вызове значение регистра CS размещается в стеке перед значением регистра PC, а счетчик команд и регистр кодового сегмента загружаются непосредственными данными или с действительного адреса. На этом исполнение команды CALL завершается.

Команда возврата, RET, выталкивает из стека адрес возврата, сохраняет его в счетчике команд, после чего исполнение программы возобновляется с команды, следующей непосредственно за командой CALL. Иногда в команде RET роль непосредственных данных исполняет положительное число. Это число рассматривается как ряд байтов аргументов, размещенных в стеке перед вызовом; оно прибавляется к значению SP, и стек очищается. При дальнем вызове (исполнении команды RETF) регистр кодового сегмента выталкивается после счетчика команд.

В рамках подпрограммы необходимо реализовать доступа к аргументам. По этой причине исполнение подпрограммы часто начинается с размещения в стеке указателя базы и копирования текущего значения регистра SP в регистр BP. Таким образом, указатель базы указывает на свое предыдущее значение. После адрес возврата определяется путем прибавления к BP двойки, а первый и второй аргументы располагаются по действительным адресам BP + 4 и BP + 6 соответственно. Если процедуре нужны локальные переменные, необходимое количество байтов можно вычесть из указателя стека; обращаться к этим переменным можно из указателя базы путем отрицательного смещения. В примере на рис. B.3 имеются три однословных локальных переменных, расположенных по адресам BP – 2, BP – 4 и BP – 6. Таким образом, весь набор текущих аргументов и локальных переменных становится доступным через регистр BP.

В стеке, как обычно, сохраняются промежуточные результаты или подготавливаются аргументы для последующих вызовов. Чтобы восстановить стек до возврата, не высчитывая, какую часть стека заняла подпрограмма, нужно скопировать указатель базы в указатель стека, вытолкнуть из стека старое значение BP и исполнить команду RET.

Иногда в процессе вызова подпрограммы значения регистров процессора меняются. В этой связи полезно сделать так, чтобы вызывающая программа не знала о том, какие регистры используются вызываемой программой. Для этого проще всего ввести для системных вызовов и обычных подпрограмм одни и те же соглашения. Предполагается, что в вызываемой программе могут изменяться значения регистров AX и DX. Если в одном из этих регистров содержится ценная информация, желательно, чтобы вызывающая процедура разместила ее в стеке до выполнения аналогичной операции с аргументами. Если подпрограмма использует и другие регистры, их можно поместить в стек непосредственно в начале ее исполнения и извлечь из него перед запуском команды RET. Таким образом, желательно сделать так, чтобы вызывающая процедура сохраняла значения регистров AX и DX (если в них содержатся значимые данные), а вызываемая — значения других регистров, которые она перезаписывает.

Системные вызовы и системные подпрограммы

Поскольку программы запускаются поверх операционной системы, при программировании на языке ассемблера не нужно самостоятельно реализовывать операции открытия, закрытия, чтения и записи файлов. Интерпретатор поддерживает семь системных вызовов и пять функций, позволяющих ему работать на разных платформах. Все они перечислены в табл. B.4.

Таблица В.4. Некоторые системные вызовы и подпрограммы UNIX, поддерживаемые интерпретатором

№	Имя	Аргументы	Возвращаемое значение	Описание
5	_OPEN	*name, 0/1/2	Дескриптор файла	Открытие файла
8	_CREAT	*name, *mode	Дескриптор файла	Создание файла
3	_READ	fd, buf, nbytes	# байт	Чтение n байт (nbytes) из буфера buf
4	_WRITE	fd, buf, nbytes	# байт	Запись n байт (nbytes) из буфера buf
6	_CLOSE	fd	0 в случае успешного исполнения	Закрытие файла с дескриптором fd
19	_LSEEK	fd, offset(long), 0/1/2	Положение (long)	Перемещение указателя файла
1	_EXIT	status		Закрытие файлов, остановка процесса
117	_GETCHAR		Чтение символа	Чтение символа в файле стандартного ввода
122	_PUTCHAR	char	Запись байта	Запись символа в файл стандартного вывода
127	_PRINTF	*format, arg		Запись с форматированием в файл стандартного вывода
121	_SPRINTF	but, *format, arg		Запись в буфере buf с форматированием
125	_SSCANF	buf, *format, arg		Чтение аргументов из буфера buf

Все эти двенадцать программ запускаются в стандартном порядке вызова: в первую очередь, в стеке в обратном порядке размещаются все необходимые аргументы; затем в стек вводится номер вызова; наконец, исполняется команда системного исключения SYS без операндов. Все необходимые данные, в том числе номер вызова той или иной системной службы, системная программа извлекает из стека. Возвращаемые значения записываются либо в регистр AX, либо в комбинацию регистров DX : AX (если они соответствуют длинному слову).

После завершения команды SYS значения всех регистров гарантированно сохраняются. В стеке после вызова остаются и аргументы. Если они не понадобятся для следующего вызова, вызывающая процедура должна скорректировать содержимое указателя стека.

Для удобства имена системных вызовов можно в начале программы определить как константы; в таком случае к ним можно будет обращаться не по номеру, а по имени. Позже мы рассмотрим несколько системных вызовов на примерах, а пока что воздержимся от излишней детализации.

Файлы открываются вызовом OPEN или CREAT. В обоих случаях первым аргументом является адрес начала строки, содержащей имя файла. Вторым аргументом в вызове OPEN могут быть 0 (если файл нужно открыть для чтения), 1 (если файл открывается для записи) или 2 (если файл открывается для чтения и записи). Если при наличии прав записи указанный файл не существует, в процессе вызова он создается. Во время вызова CREAT создается пустой файл с полномочиями, заданными на основании второго аргумента. Вызовы OPEN и CREAT возвращают двухбайтовое целое и помещают его в регистр AX, который называется **дескриптором файла** и с помощью которого этот файл можно прочесть, записать или закрыть. Отрицательное возвращенное значение свидетельствует о том, что вызов завершился ошибкой. Перед началом исполнения программы открываются три файла с дескрипторами: 0 для стандартного ввода, 1 для стандартного вывода и 2 для стандартного вывода ошибок.

У вызовов READ и WRITE три аргумента: дескриптор файла, буфер, в котором хранятся данные, и число передаваемых байтов. Поскольку аргументы размещаются в стеке в обратном порядке, сначала в стек вводится число байтов, затем — адрес начала буфера, далее — дескриптор файла, и наконец — номер вызова (READ или WRITE). Этот порядок размещения аргументов в стеке аналогичен стандартной последовательности вызова в языке C:

```
read(fd, buffer, bytes);
```

Эта команда реализуется путем введения в стек параметров в следующем порядке: bytes, buffer, fd.

У вызова CLOSE один аргумент — дескриптор файла. При успешном закрытии файла он возвращает значение 0 в регистре AX. Вызов EXIT требует сохранения в стеке статуса выхода и не возвращает значение.

Вызов LSEEK изменяет **указатель чтения-записи** в открытом файле. Первым аргументом здесь выступает дескриптор файла. Поскольку второй аргумент является длинным словом, в стек сначала помещается старшее слово, а затем младшее — даже в том случае, если смещение умещается в одном слове. Третий аргумент определяет метод вычисления нового значения указателя чтения-записи: относительно начала файла (0), текущего положения (1) или конца файла (2). Возвращаемое значение определяет новое положение указателя относительно начала файла; в формате длинного слова оно сохраняется в комбинации регистров DX : AX.

Перейдем к функциям, не являющимся системными вызовами. Функция GETCHAR считывает один символ из файла стандартного ввода и помещает его в регистр AL; при этом значение AH обнуляется. При ошибке в регистр AX записывается значение –1. Вызов PUTCHAR записывает один байт в файл стандартного вывода. Выводимым значением при успешной записи является именно этот байт; при ошибке выводится –1.

Вызов PRINTF выводит отформатированные данные. Первым аргументом служит адрес форматной строки, задающей формат выводимых данных. Строка %d указывает на то, что следующим аргументом является целое число в стеке, которое при выводе преобразуется в десятичное представление. Аналогичным образом, строка %x осуществляет преобразование в шестнадцатеричное, а %o — в восьмеричное представление. Строка %s определяет следующий аргумент как

строку с завершающим нулем, которая при вызове передается с помощью адреса в стеке. Количество дополнительных аргументов в стеке должно соответствовать количеству указателей преобразования в форматной строке.

Например:

```
printf(??x = %d and y = %d\n??, x, y);
```

Этот вызов заменяет при выводе численные значения x и y строками %d из форматной строки. В целях совместимости с языком C в стеке сначала размещается аргумент y, затем — x, и в завершение — адрес форматной строки. Это соглашение обуславливается тем, что у процедуры printf переменное число параметров, и при условии их размещения в стеке в обратном порядке форматная строка всегда остается последней. Если бы параметры размещались в стеке слева направо, форматная строка оказалась бы слишком глубоко, и процедура printf не смогла бы ее обнаружить.

Первым аргументом вызова SPRINTF является буфер, в который направляется выходная строка (в отличие от PRINTF, где строка попадает в файл стандартного вывода). Остальные аргументы аналогичны тем, что применяются в PRINTF. Вызов SSCANF в определенном смысле противоположен PRINTF — первым аргументом здесь является строка, которая может содержать целые числа в десятичном, восьмеричном или шестнадцатеричном представлении, а вторым — форматная строка с указателями преобразования. Могут быть и другие аргументы — адреса слов памяти, принимающие преобразованные данные. Все упомянутые системные подпрограммы весьма разносторонни, но подробный анализ их возможностей мы проводить не будем. В разделе «Примеры» есть несколько примеров, демонстрирующих их применение в различных ситуациях.

Заключительные замечания о наборе команд

В официальной спецификации набора команд 8088 предусмотрен префикс **перекрытия сегмента**, который позволяет применять действительные адреса из других сегментов; первый адрес в памяти, следующий за префиксом перекрытие, вычисляется при помощи указанного сегментного регистра. К примеру:

```
ESEG MOV DX,(BX)
```

Эта команда сначала вычисляет адрес BX, используя для этого дополнительный сегмент, а затем перемещает содержимое в регистр DX. В то же время ни стековый сегмент (в случае адресации с использованием регистра SP), ни дополнительный сегмент (в отношении строковых команд, оперирующих регистром DI) перекрыть нельзя. Сегментные регистры SS, DS и ES могут быть задействованы в команде MOV, но переместить непосредственные данные в сегментный регистр нельзя, в а операции XCHG эти регистры не используются. Менять сегментные регистры и практиковать перекрытие довольно сложно, и по возможности этих методов в программировании лучше избегать. Так как интерпретатор работает с фиксированными сегментными регистрами, здесь подобных проблем не возникает.

Команды исполнения операций с плавающей точкой предусмотрены в большинстве компьютеров. Иногда они выполняются в процессоре, иногда — в сопроцессоре. Бывает и так, что они лишь программно интерпретируются при помощи специального исключения. Более детально эту тему мы развивать не будем.

Ассемблер

Итак, мы закончили обсуждение архитектуры 8088. Данный раздел посвящен программному обеспечению, позволяющему программировать процессор 8088 на языке ассемблера, в частности, тем инструментальным средствам, которые мы предлагаем в качестве основы для обучения программированию на этом языке. Сначала мы рассмотрим ассемблер (программу ассемблирования), затем — трассер (программу трассировки), и, наконец, обсудим некоторые практические моменты их применения.

Введение

До настоящего момента мы обозначали команды их **мнемониками** — краткими и легкими для запоминания символическими именами, как то `ADD` или `CMP`. Регистры также назывались символическими именами — `AX`, `BP` и т. д. Программа, написанная с применением символических имен команд и регистров, называется **программой на языке ассемблера**, или просто **ассемблерной программой**. Чтобы исполнить такую программу, в первую очередь необходимо преобразовать ее в двоичные числа, с которыми работает процессор. Прикладная программа, которая осуществляет такое преобразование, называется **ассемблером**. То, что получается в результате работы этой программы (то есть в результате ассемблирования), называется **объектным файлом**. Многие программы выполняют вызовы уже ассемблированных подпрограмм, хранящихся в библиотеках. Чтобы такие программы могли исполняться, ассемблированный объектный файл и библиотечные подпрограммы, к которым он обращается (а они также существуют в виде объектных файлов), необходимо объединить в **исполняемый двоичный файл**. Эту операцию проводит специальная программа, называемая **компоновщиком**. Ассемблирование считается полностью завершенным лишь после сборки компоновщиком исполняемого двоичного файла из одного или нескольких объектных файлов. Затем операционная система помещает этот файл в память и исполняет его.

В первую очередь ассемблер должен сформировать **таблицу символических имен** (или сокращенно «таблицу символов»); с ее помощью имена символических констант и меток отображаются на двоичные числа, которым они фактически соответствуют. Константы, явно определенные в программе, можно разместить в таблице символических имен без предварительной обработки. Метки же представляют адреса, значения которых не столь очевидны. Для определения этих значений ассемблер проводит построчный анализ программы, называемый **первым проходом**. Во время этого прохода он отслеживает показания **счетчика адресов**, который обычно обозначается **точкой**. При обнаружении на первом проходе каждой команды или операции резервирования памяти счетчик адресов увеличивается на величину, которая выражает объем памяти, необходимый для размещения данного элемента. Таким образом, если первые две команды занимают 2 и 3 байта соответственно, то метка третьей команды принимает численное значение 5. К примеру, если следующий фрагмент кода находится в начале программы, значением метки `L` будет 5:

```
MOV AX,6
MOV BX,500
L:
```

В начале **второго прохода** численные значения всех символов уже известны. Так как численные значения мнемонических кодов команд постоянны, становится возможной **генерация кода**. Команды вновь считываются, одна за другой, и их двоичные значения записываются в объектный файл. После ассемблирования последней команды создается объектный файл.

Ассемблер as88 из набора ACK

В этом подразделе мы подробно обсудим ассемблер/компоновщик as88, который можно найти на англоязычном веб-сайте книги. Этот ассемблер, входящий в набор ACK (Amsterdam Compiler Kit), больше похож на ассемблеры UNIX, чем на аналогичные продукты для MS-DOS и Windows. Символом комментария в нем является знак восклицания (`!`). Все символы, следующие за знаком восклицания вплоть до конца строки, считаются комментарием и не влияют на объектный файл. Пустые строки допустимы, но при формировании объектного файла не учитываются.

В ассемблере имеются три секции, в которых хранятся транслируемый код и данные. Разделение на секции соответствует сегментации памяти. Первая секция, **секция текста**, предназначена для хранения команд процессора. Следующая секция, **секция данных**, служит для инициализации памяти в сегменте данных, который к началу процесса становится известен. Последняя секция, секция **BSS** (Block Started by Symbol — блок с начальным символом), нужна для резервирования памяти в неинициализированном (точнее говоря, в инициализированном нулем) сегменте данных. В каждой из трех секций есть собственный счетчик адресов. Секции задумывались для того, чтобы ассемблер мог сгенерировать несколько команд, затем перейти к данным, затем вернуться к командам, потом снова к данным, и т. д., а компоновщик, в свою очередь, мог скомпоновать все эти блоки так, чтобы все команды оказались вместе в текстовом сегменте, а все слова данных — в сегменте данных. Результат обработки любой отдельно взятой строки на языке ассемблера попадает только в одну из секций, хотя строки кода и строки данных могут чередоваться. На стадии исполнения секция текста хранится в текстовом сегменте, а секции данных и BSS — последовательно в сегменте данных.

Как команда, так и слово данных в программе на языке ассемблера может начинаться с метки. Метка также может быть единственным содержанием строки, и в этом случае она считается относящейся к следующей команде или слову данных. Пример:

```
    CMP AX,ABC
    JE L
    MOV AX,XYZ
L:
```

Здесь `L` — метка, связанная со следующей командой или словом. Метки бывают двух видов. Во-первых, существуют **глобальные метки**, представляющие собой буквенно-цифровые идентификаторы, после которых ставится двоеточие (`:`).

Такие метки должны быть уникальными; они не могут совпадать с ключевыми словами и мнемоническими кодами команд. Во-вторых, в секции текста встречаются **локальные метки**, каждая из которых состоит из одной цифры с последующим двоеточием (`:`). Локальная метка может устанавливаться многократно. Например:

```
JE 2f
```

Эта команда означает операцию перехода `JUMP EQUAL` вперед до следующей локальной метки 2. Аналогично:

```
JNE 4b
```

Эта команда означает операцию перехода `JUMP NOT EQUAL` назад к ближайшей локальной метке 4.

Ассемблер допускает присвоение константам символических имен в рамках синтаксической конструкции

идентификатор = выражение

Здесь *идентификатор* является буквенно-цифровой строкой, как в следующем примере:

```
BLOCKSIZE = 1024
```

Во всех идентификаторах в рассматриваемом языке ассемблера значимыми являются только первые восемь символов. Таким образом, идентификаторы `BLOCKSIZE`, `BLOCKSIZZ` и `BLOCKSIZ` идентичны. Выражения могут состоять из констант, численных значений и операторов. Поскольку численные значения меток становятся известными к завершению первого прохода, они приравниваются к константам.

Численные значения могут быть **восьмеричными** (в таком случае они начинаются с нуля), **десятеричными** или **шестнадцатеричными** (начинаются с символов 0X или 0x). В шестнадцатеричном представлении буквы a–f и A–F обозначают значения от 10 до 15. Целочисленные операторы `+`, `-`, `*`, `/` и `%` выражают сложение, вычитание, умножение, деление и остаток о деления соответственно. Логические операторы `&`, `^` и `~` обозначают поразрядное И, поразрядное ИЛИ и логическое отрицание (НЕ) соответственно. Для группировки в выражениях могут устанавливаться квадратные скобки. Во избежание путаницы круглые скобки в выражениях *не* используются (оставлены для задания режимов адресации).

К меткам в составе выражений следует относиться с осторожностью. Вычитание меток команд из меток данных недопустимо. Разность между однородными метками выражается в численных значениях, однако ни сами метки, ни их разности не могут выступать в качестве констант в мультипликативных и логических выражениях. Выражения, разрешенные к применению в определениях констант, могут быть задействованы в виде констант в командах процессора. В некоторых процессорах предусмотрены макросредства, позволяющие группировать множества команд и присваивать таким группам имена, но в as88 этой возможности нет.

В каждом языке ассемблера есть директивы, которые влияют на процесс ассемблирования, но не транслируются в двоичный код. Они называются **псевдокомандами**. Псевдокоманды ассемблера as88 перечислены в табл. В.5.

Таблица В.5. Псевдокоманды ассемблера as88

Псевдокоманда	Описание
.SECT .TEXT	Ассемблирование следующих строк в секции текста
.SECT .DATA	Ассемблирование следующих строк в секции данных
.SECT .BSS	Ассемблирование следующих строк в секции BSS
.BYTE	Ассемблирование аргументов в виде последовательности байтов
.WORD	Ассемблирование аргументов в виде последовательности слов
.LONG	Ассемблирование аргументов в виде последовательности длинных слов
.ASCII "str"	Сохранение строки str в виде ASCII-строки без завершающего нулевого байта
.ASCIZ "str"	Сохранение строки str в виде ASCII-строки с завершающим нулевым байтом
.SPACE n	Продвижение счетчика адресов на n позиций
.ALIGN n	Продвижение счетчика адресов до n-байтной границы
.EXTERN	Объявление идентификатора внешним

Псевдокоманды из первого блока формируют секцию, в которой все последующие строки обрабатываются ассемблером. Как правило, определение такой секции размещается на отдельной строке в произвольной части кода. По соображениям реализации, первой должна быть секция текста, затем — секции данных и BBS. После этих исходных ссылок секции могут следовать в произвольном порядке. Помимо прочего, в первой строке секции должна быть установлена глобальная метка. Иных ограничений на порядок следования секций не существует.

Во втором блоке псевдокоманд содержатся указатели типов данных в сегменте данных. Таких типов всего четыре: `.BYTE`, `.WORD`, `.LONG` и строка. В первых трех типах после необязательной метки и ключевого слова псевдокоманды остаток строки занимает список константных выражений с разделением запятыми. Для строк предусмотрено два ключевых слова: `ASCII` и `ASCIZ`. Единственное различие между ними заключается в том, что ключевое слово `ASCIZ` добавляет к концу строки нулевой байт. Оба ключевых слова в обязательном порядке сопровождаются строкой, заключенной в двойные кавычки. В определениях строк допускается ряд символов-заменителей, которые перечислены в табл. В.6. Вдобавок к ним любой конкретный символ может предваряться обратной косой чертой и выражаться своим восьмеричным представлением, например \377 (максимальное число символов — три, 0 в данном случае указывать не требуется).

Таблица В.6. Некоторые разрешенные в as88 символы-заменители

Символ-заменитель	Описание
\n	Новая строка (перевод строки)
\t	Табуляция
\\	Обратная косая черта

продолжение ⇩

Таблица В.6 *(продолжение)*

Символ-заменитель	Описание
\b	Пробел
\f	Подача страницы
\r	Возврат каретки
\"	Двойная кавычка

Псевдокоманда `SPACE` увеличивает значение указателя адресов на число байтов, определенное аргументами. Это ключевое слово может быть особенно полезным, если установить его после метки в секции `BSS` в целях резервирования памяти для переменной. Ключевое слово `ALIGN` позволяет продвинуть указатель адресов до первой 2-, 4- или 8-байтной границы в памяти, что упрощает ассемблирование слов, двойных слов и т. д. с размещением в подходящих ячейках памяти. Наконец, ключевое слово `EXTERN` объявляет о доступности указанной программы или ячейки памяти для компоновщика с целью установки внешних ссылок. Определение не обязательно должно находиться в текущем файле; оно может быть в любом месте в пределах досягаемости компоновщика.

Следует сделать ряд замечаний касательно совместного применения ассемблера и трассера. Ассемблер воспринимает ключевые слова как в верхнем, так и в нижнем регистрах; трассер выводит их только в верхнем регистре. Ассемблер воспринимает в качестве символа новой строки символы-заменители `\r` (возврат каретки) и `\n` (перевод строки), в то время как трассер использует только последний. Ассемблер способен работать с программами, разбитыми на несколько файлов, в то время как для обработки в трассере всю программу необходимо объединить в файл с расширением .$. Включенные в него файлы запрашиваются командой

```
#include имя_файла
```

Включаемый файл в таком случае также должен находиться на указанной позиции в рамках единого $-файла. Ассемблер проверяет, была ли ранее проведена обработка данного включаемого файла, и загружает одну его копию. Эта возможность особенно полезна в тех случаях, когда один заголовочный файл является общим для нескольких файлов. В такой ситуации в объединенный исходный файл включается только одна его копия. Для включения файла команда `#include` должна быть первым маркером строки без предшествующих разделителей, а путь к файлу должен быть заключен в двойные кавычки.

При наличии одного исходного файла (например, pr.s) предполагается, что именем проекта являются символы *pr*, а объединенный файл должен называться pr.$. Если исходных файлов несколько, основа имени первого из них принимается за имя проекта, и она же применяется для определения $-файла, который генерируется ассемблером путем объединения исходных файлов. Этот механизм можно изменить, поместив в командной строке перед первым исходным файлом флаг `-o projname`; в таком случае объединенный файл получает имя projname.$.

Имейте в виду, что к применению включаемых файлов и нескольких исходных файлов выдвигается ряд условий. В частности, во всех исходных файлах имена меток, переменных и констант должны быть разными. Более того, в конечном

счете, в загрузочный файл ассемблируется файл projname.$, поэтому номера строк, указываемые ассемблером при обнаружении ошибок и выводе предупреждений, отсчитываются именно от этого файла. При работе с небольшими проектами имеет смысл писать программу в одном файле, а не использовать команду `#include`.

Некоторые отличия от других ассемблеров 8088

Ассемблер as88 построен по модели стандартного ассемблера UNIX, в связи с чем он в некоторых отношениях довольно существенно отличается от макроассемблера Microsoft MASM и ассемблера Borland 8088 TASM. И MASM, и TASM были разработаны для операционной системы MS-DOS, и связь операционной системы и ассемблера в определенных аспектах прослеживается весьма отчетливо. MASM и TASM поддерживают все модели памяти 8088, поддерживаемые MS-DOS. К примеру, в них предусмотрены **миниатюрная** модель памяти, в которой весь код и все данные, вместе взятые, должны умещаться в 64 Кбайт, **малая** модель, где по 64 Кбайт отводится под каждый из этих сегментов, и **большая** модель, допускающая наличие множества кодовых сегментов и сегментов данных. Различия между этими моделями зависят от способа применения сегментных регистров. В большой модели можно выполнять дальние вызовы и изменять регистр DS. Процессор сам по себе налагает некоторые ограничения на сегментные регистры (например, регистр CS не может быть целевым адресом в команде `MOV`). Для упрощения трассировки в as88 применяется модель памяти, сходная с малой, хотя ассемблер и без помощи трассера может обрабатывать сегментные регистры без каких-либо дополнительных ограничений.

В двух вышеупомянутых ассемблерах нет секции `BSS`, а инициализируют память они только в секциях данных. Обычно код на языке ассемблера начинается с заголовка в том или ином виде; затем следует секция данных, обозначаемая ключевым словом `.data`, после чего пишется текст программы с ключевым словом `.code`. В заголовке используются ключевые слова `title` (название программы), `.model` (модель памяти) и `.stack` (резервирование памяти для стекового сегмента). Если целевой двоичный файл должен быть записан с расширением .com, применяется миниатюрная модель памяти; при этом все сегментные регистры уравниваются, а в начале объединенного сегмента 256 байт резервируются для «префикса сегмента программы».

Вместо директив `.WORD` `.BYTE` и `ASCIZ` в ассемблерах TASM и MASM применяются ключевые слова `DW` (определение слова) и `DB` (определение байта). После директивы `DB` может быть помещено определение строки в двойных кавычках. После меток в определениях данных двоеточия не ставятся. Крупные блоки памяти инициализируются ключевым словом `DUP`; перед ним указывается число байтов, а после него определяется инициализация. Например:

```
LABEL DB 1000 DUP (0)
```

Этот оператор инициализирует 1000 байт памяти байтами ASCII-нулей по адресу метки `LABEL`.

После меток подпрограмм устанавливается не двоеточие, а ключевое слово `PROC`. В конце подпрограммы эта метка дублируется, а после нее ставится клю-

чевое слово ENDP, которое позволяет ассемблеру определить точную область действия подпрограммы. Локальные метки не поддерживаются.

Ключевые слова, применяемые с командами, идентичны в MASM, TASM и as88. Кроме того, в командах с двумя операндами исходный операнд ставится после целевого. С другой стороны, чаще всего для передачи аргументов функциям используется не стек, а регистры. Если же программы на языке ассемблера вставляются в программы на C или C++, стек предпочтительнее, поскольку он согласуется с механизмом вызова подпрограмм C. Это не очень принципиально, поскольку в as88 стек можно заменить регистрами.

Самое существенное различие между MASM, TASM и as88 сводится к механизму выполнения системных вызовов. В MASM и TASM они совершаются с помощью системного прерывания INT. Самый распространенный его вариант — INT 21H; он применяется для выполнения системных вызовов MS-DOS. Номер вызова при этом помещается в регистр AX (вновь аргументы передаются регистрам). Различным устройствам соответствуют разные векторы и номера прерываний, например INT 16H для клавиатурных функций BIOS и INT 10H для дисплея. Чтобы программировать эти функции, программист должен знать огромное количество данных о тех или иных устройствах. Системные вызовы UNIX, реализованные в as88, значительно проще.

Трассер

Рассматриваемый трассер/отладчик адаптирован для обычного (VT100) терминала 24 × 80 со стандартным для терминалов набором ANSI-команд. В машинах UNIX и Linux этим требованиям обычно отвечает эмулятор терминала в системе X-window. В машинах Windows для этой цели с файлами инициализации системы загружается драйвер ansi.sys; процедура загрузки которого будет описана далее. Структуру окна трассера мы уже показывали. Как видно из рис. B.4, экран трассера разделен на семь секций.

В верхней левой части окна находится секция процессора; в ней содержимое регистров общего назначения указывается в десятичном представлении, всех остальных регистров — в шестнадцатеричной. Поскольку численное значение счетчика команд не слишком информативно, строкой ниже определяется положение в исходном коде программы по отношению к предыдущей глобальной метке. Над полем счетчика команд показаны пять кодов условий. Переполнение обозначается символом *v*, флаг направления — символом > (увеличение) или < (уменьшение). Флаг знака может быть выражен символами *n* (отрицательные значения) или *p* (нуль и положительные значения). Установленный нулевой флаг обозначается как *z*, а установленный флаг переноса — как *c*. Знак – означает сброшенный флаг.

В верхней средней секции размещается стек в шестнадцатеричном представлении. Положение указателя стека обозначается стрелкой (=>). Адреса возврата подпрограмм указываются цифрой, устанавливаемой перед шестнадцатеричным значением. В верхней правой секции выводится часть исходного файла, в которой находится следующая в порядке исполнения команда. Положение счетчика команд, как и положение указателя стека, обозначается стрелкой (=>).

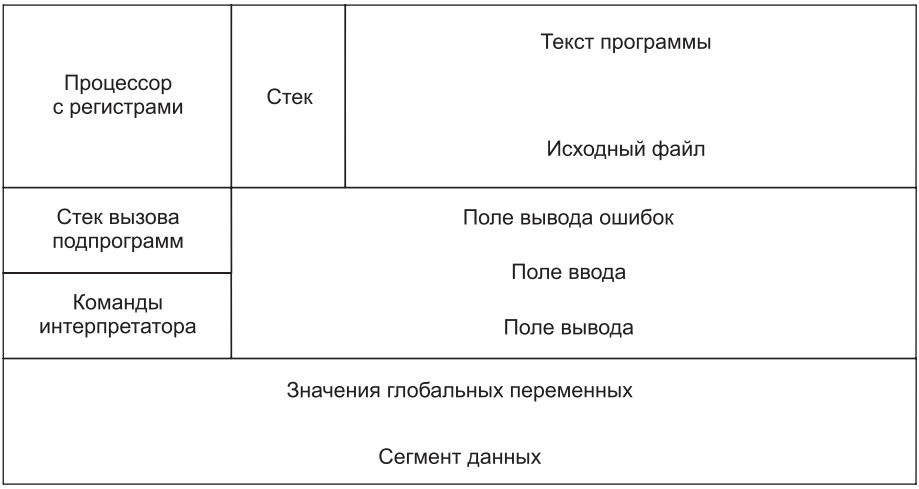


Рис. В.4. Секции окна трассера

Под секцией процессора указываются последние точки вызова подпрограмм в исходном коде. Еще ниже находится секция команд трассера, где предыдущая команда указывается сверху, а курсор команды — снизу. Имейте в виду, что каждая команда должна завершаться символом возвратом каретки (на клавиатурах ПК он вводится нажатием клавиши Enter).

В нижней секции могут находиться шесть элементов глобальной памяти данных. Каждый такой элемент начинается с позиции, отсчитываемой относительно той или иной метки, за которой следует абсолютная позиция в сегменте данных. Далее ставится двоеточие и выводится 8 байт в шестнадцатеричном представлении. Следующие 11 позиций зарезервированы для символов, за которыми могут следовать четыре слова в десятичном представлении. Байты, символы и слова представляют одну и ту же область памяти, но для символьного выражения предусмотрено три дополнительных байта. Это решение обуславливается тем, что изначально не ясно, в каком виде будут представлены данные: в виде целых чисел со знаком или без знака либо в виде строки.

Средняя правая секция отводится под ввод и вывод. В первой строке следует вывод ошибок трассера, вторую строку занимает ввод, а несколько последующих — вывод. Выводу ошибок предшествует буква *E*, вводу — буква *I*, стандартному выводу — символ >. В поле ввода стрелка (->) обозначает следующий в порядке чтения указатель. При вызовах `read` или `getchar` следующее введенное в командной строке трассера выражение попадает в поле ввода. В этом случае необходимо завершить ввод нажатием клавиши Enter. Необработанная на данный момент часть строки находится после стрелки (->).

Как правило, трассер считывает команды и входные данные из файла стандартного ввода. В то же время можно подготовить файл с командами трассера и файл со строками ввода, которые будут считаны до передачи управления файлу стандартного ввода. Файлы с командами трассера сохраняются с расширением .t, а файлы ввода — с расширением .i. В языке ассемблера в ключевых словах, системных подпрограммах и псевдокомандах могут быть задействованы символы

как в верхнем, так и в нижнем регистрах. В процессе ассемблирования создается файл с расширением .$, в котором ключевые слова в нижнем регистре преобразуются в верхний регистр, а символы возврата каретки отбрасываются. При такой системе в каждом проекте (предположим, он называется *pr*) может быть до шести файлов:

1. Файл с исходным кодом на языке ассемблера (pr.s).
2. Файл с объединенным исходным кодом (pr.$).
3. Файл загрузки (pr.88).
4. Предустановленный файл стандартного ввода (pr.i).
5. Предустановленный файл с командами трассера (pr.t).
6. Файл для компоновки кода на языке ассемблера с файлом загрузки (pr.#).

Содержимым последнего файла трассер заполняет верхнюю правую секцию окна и поле счетчика команд. Кроме того, трассер проверяет, когда был создан файл загрузки: до последнего изменения исходного кода программы или после; в первом случае генерируется предупреждение.

Команды трассера

Команды трассера перечислены в табл. В.7. Наиболее важными из них считаются команда однократного возврата (первая строка таблицы), которая исполняет одну команду процессора, и команда выхода q (нижняя строка таблицы). Число в качестве имени команды обозначает количество команд процессора, подготовленных к исполнению. Так, число *k* эквивалентно *k*-кратному повторению команды возврата. Аналогичный эффект достигается в том случае, если после числа ставится знак восклицания (!) или символ X.

Таблица В.7. Команды трассера

Адрес	Команда	Пример	Описание
			Исполнение одной команды
#	,! ,X	24	Исполнение # команд
/T +#	g , ! ,	/start +5g	Прогон до строки # после метки T
/T +#	b	/start +5b	Размещение контрольной точки в строке # после метки T
/T +#	c	/start +5c	Удаление контрольной точки со строки # после метки T
#	g	108g	Исполнение программы до строки #
	g	g	Исполнение программы до повторного достижения текущей строки
	b	b	Размещение контрольной точки в текущей строке
	c	c	Удаление контрольной точки из текущей строки
	n	n	Исполнение программы до следующей строки
	r	r	Исполнение до контрольной точки или до конца

Адрес	Команда	Пример	Описание
	=	=	Прогон программы до аналогичного уровня подпрограмм
	-	-	Прогон до уровня подпрограмм минус 1
	+	+	Прогон до уровня подпрограмм плюс 1
/D +#		/buf + 6	Отображение сегмента данных в метке +#
/D +#	d , !	/buf + 6d	Отображение сегмента данных в метке +#
	R , CTRL L	R	Обновление окон
	q	q	Завершение трассировки, возврат в командную оболочку

Команда `g` позволяет перейти к определенной строке исходного файла. Эта команда существует в трех вариантах. Если перед ней указывается номер строки, трассер продолжает работу до достижения этой строки. При наличии метки `/T` (с или без символов `+#` в адресе) номер строки, на которой трассер должен остановиться, вычисляется на основе метки команды `T`. Если перед командой `g` нет никаких дополнительных элементов, трассер продолжает исполнять команды до повторного достижения строки с текущим номером.

Существует два варианта команды `/label`: для меток команд и меток данных. В первом случае строка в нижнем окне заполняется или заменяется набором данных, начинающемся с данной метки. Во втором случае команда `/label`: эквивалентна команде `g`. После метки может быть установлен знак плюс и число (в табл. B.7 числа обозначены символом `#`), позволяющее выполнить смещение от метки.

Командой `b` можно установить **контрольную точку**. Перед командой `b` можно поставить метку команды со смещением или без него. Если во время исполнения встречается строка с контрольной точкой, трассер останавливается. Чтобы возобновить исполнение с контрольной точки, нужна команда возврата или запуска. Если метка и число опускаются, контрольная точка устанавливается в текущей строке. Снять контрольную точку позволяет специальная команда `c`, которая, подобно команде `b`, может предваряться метками и числами. Существует также команда запуска `r`, которую трассер исполняет вплоть до контрольной точки, вызова команды завершения или конца команд.

Кроме того, трассер отслеживает уровень подпрограмм, на котором работает программа. Этот уровень указывается под секцией процессора; его можно также определить по цифрам в секции стека. На уровнях подпрограмм основываются три команды. Команда `-` заставляет программу трассировки работать до того момента, пока текущий уровень подпрограмм не сменится более низким (точнее говоря, следующим в порядке понижения). Фактически эта команда продолжает исполнять команды процессора до завершения текущей подпрограммы. Противоположную функцию выполняет команда `+`, заставляющая программу трассировки работать до перехода на более высокий (следующий в порядке повышения) уровень подпрограмм. Команда `=` продолжает работу трассера до уровня, аналогичного текущему, и может применяться для исполнения подпрограмм в рамках команды `CALL`. При использовании команды `=` подробные сведения

о подпрограмме в окне трассера не указываются. Существует схожая команда n, которая продолжает исполнение до следующей строки программы. Она особенно полезна в случае вызова в качестве команды LOOP; исполнение прекращается одновременно с завершением цикла.

Подготовительные действия

В этом разделе перечислены действия, необходимые, чтобы подготовиться к работе с вышеописанными инструментами. В первую очередь нужно подобрать программное обеспечение для конкретной платформы. Мы скомпилировали версии для Solaris, UNIX, Linux и Windows. Все эти версии есть на сопроводительном компакт-диске и в Интернете по адресу www.prenhall.com/tanenbaum. Перейдите на сайт, затем — в раздел Companion Web Site, относящийся к этой книге, и, наконец, выберите нужную ссылку в левом меню. Распакуйте выбранный zip-файл в папке assembler. В этой папке и в ее вложенных папках содержится весь необходимый материал. На компакт-диске основными папками являются Bigendnx, LtlendNx и MSWindos. В каждой из них есть вложенная папка assembler, в которой, опять же, можно найти все необходимое. Три упомянутых папки предназначены для систем UNIX с прямым порядком следования байтов (то есть для рабочих станций Sun), для систем UNIX с обратным порядком следования байтов (ОС Linux, установленных на ПК) и систем Windows.

После распаковки или копирования в папке assembler должны оказаться следующие вложенные папки: READ_ME, bin, as_src, trce_src, examples и exercise. Предварительно скомпилированные исходные файлы размещены в папке bin, а соответствующие двоичные файлы — в папке examples.

Чтобы получить базовые сведения о работе системы, перейдите в папку examples и введите команду

```
t88 HlloWrld
```

Эта команда представлена в первом примере из раздела «Примеры».

Исходный код для ассемблера находится в папке as_src. Файлы исходного кода написаны на языке C, а перекомпилировать их можно командой make. Для POSIX-совместимых платформ в папке исходных файлов предусмотрена утилита Makefile, которая выполняет перекомпиляцию. Для Windows имеется командный файл make.bat. Возможно, после компиляции придется либо переместить исполняемые файлы в папку программы, либо изменить переменную PATH таким образом, чтобы сделать ассемблер as88 и программу трассировки t88 видимыми из папок с исходным кодом. Если этого не сделать, то вместо команды t88 придется вводить полный путь к файлу.

В системах Windows 2000 и XP необходимо установить драйвер терминала ansi.sys; для этого в конфигурационный файл config.nt следует добавить строку:

```
device=%systemRoot%\System32\ansi.sys
```

Этот файл располагается по следующему пути:

- в Windows 2000 — \winnt\system32\config.nt;
- в Windows XP — \windows\system32\config.nt.

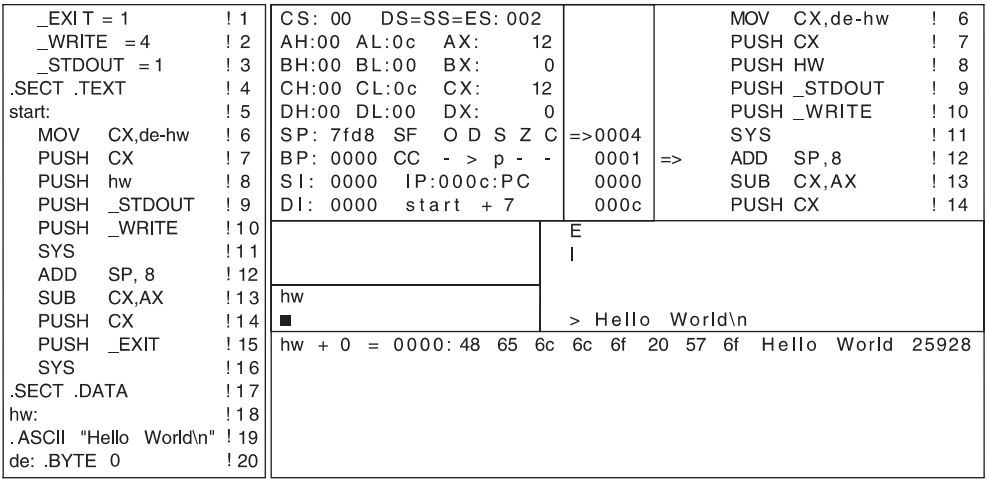
В UNIX и Linux обычно используется стандартный драйвер.

Примеры

В разделах «Процессор 8088», «Память и адресация» и «Набор команд 8088» мы рассматривали процессор 8088, его память и команды. Затем, в разделе «Ассемблер» мы разбирали основной для данного руководства язык ассемблера — as88. Раздел «Трассер» был посвящен изучению трассера. Наконец, в разделе «Подготовительные действия» были приведены инструкции по настройке набора инструментов. Теоретически, всей этой информации вполне достаточно для написания и отладки программ на языке ассемблера посредством указанных инструментальных средств. В то же время, нам кажется, что читателю будет небезынтересно познакомиться с подробными примерами программ на языке ассемблере и способами их отладки с помощью трассера. Такие примеры представлены в данном разделе. Все программы, которые мы здесь рассмотрим, можно найти в папке examples набора инструментов. Самостоятельные ассемблирование и трассировка каждого примера горячо приветствуются.

Hello World

Начнем с примера программы HlloWrld.s. В листинге B.1 представлен исходный код программы, а на рис. B.5 показано содержимое окна трассера. В листинге символ комментария (!) отделяет команды от номеров строк. В первых трех строках содержатся определения констант, привязывающие условные имена двух системных вызовов и файл вывода к соответствующим внутренним представлениям.



а *б*

Рис. B.5. Содержимое окна трассера при выполнении программы из листинга B.1

Листинг B.1. Ассемблерный код программы HlloWrld.s

```
    _EXI T = 1          ! 1
    _WRITE = 4          ! 2
    _STDOUT = 1         ! 3
.SECT .TEXT             ! 4
```

продолжение ⇩

Листинг B.1 *(продолжение)*

```
start:                  ! 5
    MOV CX,de-hw        ! 6
    PUSH CX             ! 7
    PUSH hw             ! 8
    PUSH _STDOUT        ! 9
    PUSH _WRITE         ! 10
    SYS                 ! 11
    ADD SP, 8           ! 12
    SUB CX,AX           ! 13
    PUSH CX             ! 14
    PUSH _EXIT          ! 15
    SYS                 ! 16
.SECT .DATA             ! 17
hw:                     ! 18
.ASCII "Hello World\n"  ! 19
de: .BYTE 0             ! 20
```

Псевдокоманда `.SECT` в строке 4 указывает на то, что нижеследующие строки входят в секцию текста, иначе говоря, являются командами процессора. Аналогичным образом, все, что следует за строкой 17, считается данными. В строке 19 инициализируется строка данных, состоящая из 12 байт, в том числе одного пробела и символа перевода строки (`\n`) в конце.

В строках 5, 18 и 20 содержатся метки, обозначенные двоеточием (`:`). Они представляют численные значения, схожие с константами. В данном случае ассемблеру приходится определять эти численные значения. Поскольку метка `start` находится в начале секции текста, соответствующее значение принимается за 0, но значения в последующих метках секции текста (в этом примере они не показаны) обуславливаются количеством предшествующих байтов кода. Теперь рассмотрим строку 6. Она заканчивается разностью двух меток — численной константой. Таким образом, строку 6 можно приравнять к следующему выражению:

```
MOV CX,12
```

Разница между ними заключается лишь в том, что в одном случае длину строки определяет ассемблер, а в другом — программист. Указанное здесь значение выражает объем пространства в секции данных, зарезервированное для размещения строки, расположенной в строке 19. Команда `MOV` в строке 6 представляет собой команду копирования `de –hw` в регистр `CX`.

Содержание строк 7–11 демонстрирует механизм формирования системных вызовов в применяемом наборе инструментов. Фактически, эти строки представляют собой переведенный на языке ассемблера вызов функции из языка C:

```
write(1, hw, 12);
```

Здесь первый параметр представляет собой дескриптор файла стандартного вывода (1), второй выражает адрес отображаемой строки (`hw`), а третий — длину строки (12). В строках 7–9 эти параметры помещаются в стек в обратном порядке, что соответствует последовательности вызова, принятой в C и применяемой данной программой трассировки. В строке 10 в стек вводится номер системного вызова для функции `write` (4), а в строке 11 выполняется сам вызов. Такой порядок по большей части соответствует механизму исполнения программы на языке ассемблера в клонах UNIX (или Linux), однако при работе в другой операцион-

ной системе его необходимо скорректировать в расчете на конкретные правила выполнения системных вызовов. Впрочем, даже при работе в среде Windows ассемблер as88 и трассер t88 реализуют правила вызовов, принятые в UNIX.

Системный вызов в строке 11 ответственен за вывод данных. Код в строке 12 выполняет очистку стека, возвращая указатель стека к значению, которое у него было до размещения в стеке четырех 2-байтных слов. Если вызов `write` проводится успешно, число записанных байтов возвращается в регистре AX. В строке 13 результат системного вызова после строки 11 вычитается из исходной длины строки, записанной в CX; тем самым производится проверка успешности вызова, то есть фактической записи всех байтов. Таким образом, код завершения должен быть равен нулю при успешном вызове и, соответственно, не равен нулю в противном случае. В строках 14 и 15 подготавливается системный вызов `exit`, осуществляемый в строке 16; для этого коды завершения и функции, относящиеся к вызову `EXIT`, отправляются в стек.

Имейте в виду, что в командах `MOV` и `SUB` первый аргумент указывает на приемник, а второй — на источник. Такова особенность нашего ассемблера; в других ассемблерах порядок может быть иным. Выбор разработчиками того или иного варианта, по большому счету, произволен.

Теперь попробуем ассемблировать и запустить программу HlloWrld.s. Представляемые команды подходят как для UNIX, так и для Windows. В средах Linux, Solaris, MacOS X и других клонах UNIX процедура аналогична той, что используется для базовой версии UNIX. Во-первых, откройте окно командной строки (командную оболочку). В Windows в большинстве случаев для этого следует выбрать команду Пуск▸Все программы▸Стандартные▸Командная строка (Start▸Programs▸Accessories▸Command Prompt). Далее, перейдите к каталогу examples с помощью команды `cd`. Аргумент этой команды выбирается в зависимости от местоположения набора инструментов в файловой системе. Затем проверьте, имеются ли в этом каталоге двоичные файлы ассемблера и трассера; для этого воспользуйтесь командой `ls` (UNIX) или `dir` (Windows). Эти файлы называются as88 и t88 соответственно. В среде Windows они имеют расширение .exe, но в командах его указывать не нужно. Если файлы ассемблера и трассера в названном каталоге отсутствуют, найдите их и скопируйте в него.

После этого выполните ассемблирование тестовой программы с помощью команды

```
as88 HlloWrld.s
```

Если двоичный файл ассемблера действительно находится в каталоге examples, но после запуска этой команды выводится сообщение об ошибке, в UNIX попробуйте ввести строку:

```
./as88 HlloWrld.s
```

В Windows для той же цели используйте строку:

```
.\as88 HlloWrld.s
```

В случае успешного завершения ассемблирования должны быть выведены следующие сообщения:

```
Project HlloWrld listfile HlloWrld.$
Project HlloWrld num file HlloWrld.#
Project HlloWrld loadfile HlloWrld.88
```

Естественно, должны быть также созданы соответствующие файлы. Если никаких сообщений об ошибках не было, введите команду

```
t88 HlloWrld
```

В результате в верхней правой секции окна трассера появится стрелка, указывающая на команду

```
MOV CX,de-hw
```

Это команда из строки 6 листинга B.1. После этого нажмите клавишу возврата каретки (на клавиатурах ПК она называется Enter). Как видите, теперь стрелка указывает на команду

```
PUSH CX
```

В регистре CX, согласно содержанию левой секции окна, теперь находится значение 12. Еще раз нажмите клавишу возврата каретки и обратите внимание, что в средней верхней секции появилось значение 000c — шестнадцатеричный аналог десятичного числа 12. В этой секции демонстрируется содержимое стека, где в данный момент находится одно слово — 12. Нажмите клавишу возврата каретки еще три раза и ознакомьтесь с тем, как будут обрабатываться команды из строк 8–10. После этого в стеке должно быть четыре элемента, а в левой секции в качестве значения счетчика команд указано шестнадцатеричное число 000b.

При следующем нажатии клавиши возврата каретки будет исполнен системный вызов, а в правой нижней секции окна появится строка:

```
"Hello World\n"
```

Как видите, теперь значением регистра SP является 0x7ff0. Еще одно нажатие клавиши приведет к увеличению SP на 8 — до 0x7ff8. Через четыре нажатия клавиши возврата каретки системный вызов `exit` завершится, как и сама трассировка.

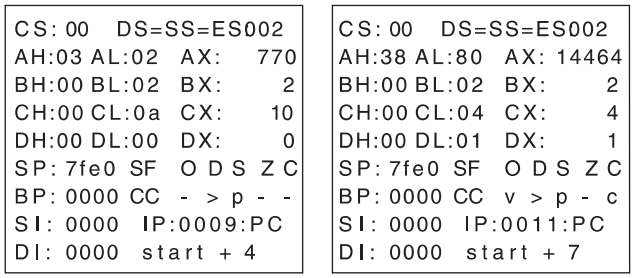
Чтобы понять, как это все работает, полезно открыть файл hlloWrld.s в любом текстовом редакторе (от применения текстовых процессоров в данном случае лучше отказаться). В UNIX это можно сделать в ex, vi или emacs; в Windows выбор естественным образом падает на Блокнот (Notepad), для вызова которого обычно следует выбрать команду Пуск▸Все программы▸Стандартные▸Блокнот (Start▸Programs▸Accessories▸Notepad).

Текстовый процессор Word не годится, так как форматирование может исказить текст программы.

Измените сообщение в строке 19, сохраните файл, ассемблируйте его и запустите в трассере. Так вы сделаете первый шаг в области программирования на языке ассемблера.

Регистры общего назначения

Следующий пример в подробностях демонстрирует механизм отображения регистров, а также один из «подводных камней» операции умножения по версии процессора 8088. Часть программы genReg.s, начиная с метки `start`, представлена в листинге B.2, а на рис. B.6 показана секция регистров окна трассера после исполнения строки 7.



а *б*

Рис. В.6. Секция регистров окна трассера: после исполнения строки 7 (*а*); после семи проходов цикла умножения (*б*)

Листинг В.2. Часть программы genReg.s

```
start:               ! 3
    MOV AX,258       ! 4
    ADDB AH,AL       ! 5
    MOV CX,(times)   ! 6
    MOV BX,muldat    ! 7
    MOV AX,(BX)      ! 8
llp: MUL 2(BX)       ! 9
    LOOP llp         ! 10
.SECT .DATA          ! 11
times: .WORD 10      ! 12
muldat: .WORD 625,2  ! 13
```

Следующая команда в строке 4 загружает в регистр AX значение 258:

```
MOV AX,258
```

В результате регистр AH получает значение 1, а регистр AL — 2. Далее, в строке 5, значения AL и AH суммируются, и значение AH оказывается равным трем. В строке 6 в CX копируется содержимое переменной `times` (10). В строке 7 в регистр BX загружается адрес переменной `muldat`, равный 2, поскольку она находится во втором байте секции данных. Именно в этот момент сделан снимок экрана, представленный на рис. В.6, *а*. Следует иметь в виду, что значение AH составляет 3, значение AL — 2, а AX — 770; в этом нет ничего удивительного, так как $3 \times 256 + 2 = 770$.

Следующая команда (в строке 8) копирует содержимое переменной `muldat` в регистр AX. Таким образом, после нажатия клавиши возврата каретки значение AX изменится на 625.

Теперь мы готовы запустить цикл, призванный умножить содержимое регистра AX на слово `2BX` (то есть `muldat` + 2), в котором находится значение 2. Неявным целевым адресом для команды `MUL` является комбинация регистров DX : AX. В первом проходе цикла результат (1250) умещается в одно слово, поэтому он сохраняется в регистре AX, а значение DX остается нулевым. Содержимое всех регистров после 7 проходов цикла умножения показано на рис. В.6, *б*.

Так как исходным значением AX являлось 625, после семи операций умножения на два результат становится равным 80 000. Поскольку он не помещается в регистр AX, произведение сохраняется в 32-разрядном регистре, образуемом путем конкатенации регистров DX : AX; таким образом, в DX содержится значение 1,

а в AX — 14 464. При вычислении действительно получается, что 1 × 65 536 + + 14 464 = 80 000. Обратите внимание, что значением CX на данном этапе является 4, поскольку команда LOOP уменьшает его на единицу во время каждого прохода. Так как исходным значением этого регистра было 10, после семи вызовов команды MUL (и шести проходов команды LOOP) значение CX уменьшилось до 4.

В ходе следующей операции умножения возникают трудности. При умножении задействуется только значение AX, в то время как значение DX игнорируется; таким образом, команда MUL умножает AX (14 464) на 2 и получает 28 928. То есть новым значением AX становится 28 928, а регистр DX обнуляется, что численно неверно.

Вызов регистров команд и указателя

Наш следующий пример — vecprod.s — представляет собой небольшую программу, вычисляющую внутреннее произведение двух векторов: vec1 и vec2. Ее код представлен в листинге В.3.

Листинг В.3. Программа vecprod.s

```
_EXIT    = 1                        ! 1 определение значения _EXIT
_PRINTF    = 127                    ! 2 определение значения _PRINTF
.SECT .TEXT                         ! 3 начало секции текста
inpstart:                           ! 4 определение метки inpstart
    MOV BP,SP                       ! 5 сохранение SP в BP
    PUSH vec2                       ! 6 введение в стек адреса vec2
    PUSH vec1                       !   введение в стек адреса vec1
    MOV CX,vec2-vec1                ! 8 CX = число байтов в векторе
    SHR CX,1                        ! 9 CX = число слов в векторе
    PUSH CX                         ! 10 введение в стек числа слов
    CALL vecmul                     ! 11 вызов vecmul
    MOV (inprod),AX                 ! 12 перемещение AX
    PUSH AX                         ! 13 введение в стек
                                    !    выводимого результата
    PUSH pfmt                       ! 14 введение в стек
                                    !    адреса форматной строки
    PUSH _PRINTF                    ! 15 введение в стек
                                    !    кода функции PRINTF
    SYS                             ! 16 вызов функции PRINTF
    ADD SP,12                       ! 17 очистка стека
    PUSH 0                          ! 18 введение в стек кода состояния
    PUSH _EXIT                      ! 19 введение в стек кода функции EXIT
    SYS                             ! 20 вызов функции EXIT
vecmul:                             ! 21 начало vecmul(count, vec1, vec2)
    PUSH BP                         ! 22 введение в стек значения BP
    MOV BP,SP                       ! 23 копирование SP в BP
                                    !    для доступа к аргументам
    MOV CX,4(BP)                    ! 24 помещение счетчика в CX
                                    !    для управления циклом
    MOV SI,6(BP)                    ! 25 SI = vec1
    MOV DI,8(BP)                    ! 26 DI = vec2
    PUSH 0                          ! 27 введение в стек значения 0
1:  LODS                            ! 28 перемещение (SI) в AX
    MUL (DI)                        ! 29 умножение AX на (DI)
    ADD -2(BP),AX                   ! 30 прибавление AX
```

```
                                    !    к накопленному в памяти значению
    ADD DI,2                        ! 31 приращение DI для указания
                                    !    на следующий элемент
    LOOP 1b                         ! 32 если CX > 0, возврат к метке 1b
    POP AX                          ! 33 выталкивание вершины стека в AX
    POP BP                          ! 34 восстановление BP
    RET                             ! 35 возврат из подпрограмы

.SECT .DATA                         ! 36 начало секции данных
pfmt: .ASCIZ "Inner product is: %d\n" ! 37 определение строки
.ALIGN 2                            ! 38 принудительная четность адреса
vec1:.WORD 3,4,7,11,3               ! 39 вектор 1
vec2:.WORD 2,6,3,1,0                ! 40 вектор 2
.SECT .BSS                          ! 41 начало секции BSS
inprod:    .SPACE2                  ! 42 выделение пространства для inprod
```

Первая часть этой программы призвана подготовить вызов функции `vecmul`; для этого SP сохраняется в BP, а затем адреса `vec2` и `vec1` вводятся в стек, что обеспечивает функции `vecmul` возможность доступа к ним. Далее, в строке 8 длина вектора в байтах загружается в CX. После смещения этого результата на 1 бит вправо (в строке 9) значение CX выражает число слов в векторе, которое помещается в стек в строке 10. Вызов `vecmul` выполняется в строке 11.

Стоит, опять же, отметить, что аргументы подпрограмм помещаются в стек в обратном порядке, и делается это в целях соответствия принятому в языке C порядку вызова. Так, средствами C функцию `vecmul` можно вызвать следующим образом:

```
vecmul(count, vec1, vec2)
```

В ходе исполнения команды `CALL` адрес возврата помещается в стек. Путем трассировки можно определить, что этот адрес — 0x0011.

Первая команда в подпрограмме — `PUSH`. Она исполняется для указателя базы (BP) в строке 22. Значение BP сохраняется в связи с тем, что этот регистр понадобится для адресации аргументов и локальных переменных данной подпрограммы. Далее, в строке 23 в регистр BP копируется указатель стека; таким образом, новое значение указателя базы задает прежнее значение указателя стека.

После этого все готово к загрузке аргументов в регистры и резервированию пространства под локальную переменную. В следующих трех строках аргументы по одному извлекаются из стека и размещаются в регистре. Как вы помните, стек оптимизирован для хранения слов, значит, адреса должны быть четными. Адрес возврата следует сразу за прежним указателем базы, а потому обращение к нему производится как к `2(BP)`. Следующим идет аргумент `count` — `4(BP)`. Он загружается в регистр CX в строке 24. В строках 25 и 26 векторы `vec1` и `vec2` загружаются в регистры SI и DI соответственно. Для сохранения промежуточного результата данной подпрограмме нужна одна локальная переменная с исходным значением 0. В связи с этим в строке 27 в стек вводится значение 0.

Состояние процессора непосредственно перед первым проходом цикла, начинающемся в строке 28, показано на рис. В.7. В узком окне в середине верхней части (справа от регистров) изображена область стек. На его дне находится адрес `vec2` (0x0022); далее, в порядке восхождения, следуют адрес `vec1` (0x0018) и третий аргумент, выражающий число элементов в каждом векторе (0x0005). Затем указывается адрес возврата (0x0011). Цифра 1 слева от этого адреса свидетельствует о том, что он является адресом возврата, отстоящим на один уровень от

основной программы. В окне под регистрами также показана цифра 1, но на этот раз она выражает символический адрес. Выше адреса возврата в стеке следуют старое значение BP (0x7fc0) и нуль, помещаемый в стек в строке 27. Стрелка, указывающая на это значение, отражает положение указателя стека (регистра SP). В окне справа от секции стека показан фрагмент текста программы; стрелка здесь указывает на следующую команду в порядке исполнения.

Теперь рассмотрим цикл, начинающийся в строке 28. Команда `LODS` через регистр SI загружает слово памяти из сегмента данных в AX. Так как флаг направления установлен, команда `LODS` выполняется в автоинкрементном режиме, а значит, после ее завершения регистр SI указывает на следующий элемент `vec1`.

Чтобы наглядно представить происходящее, выполните команду трассера

```
t88 vecprod
```

При появлении окна трассера введите следующую команду:

```
/vecmul+7b
```

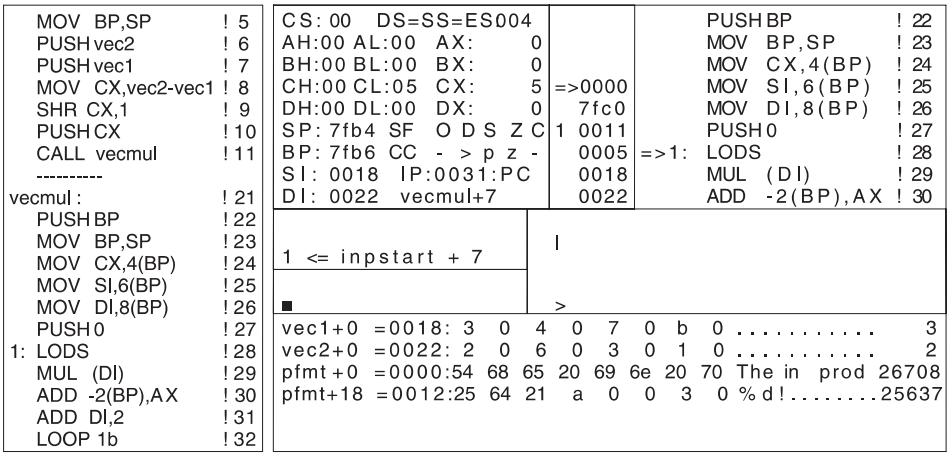


Рис. В.7. Содержимое окна трассера для программы vecprod.s при достижении строки 28, но до начала цикла

Затем нажмите клавишу возврата каретки, установив тем самым контрольную точку в строке, содержащей команду `LODS`. (Далее по тексту мы не будем напоминать о том, что после всех команд необходимо нажимать клавишу возврата каретки.) Введите команду

```
g
```

В результате выполнения этой команды трассер будет исполнять команды до контрольной точки. В данном случае он остановится в строке, содержащей команду `LODS`.

В строке 29 значение AX умножается на исходный операнд. Слово в памяти, связанное с командой `MUL`, выбирается из сегмента данных с помощью регистра DI методом косвенной регистровой адресации. Неявным (не указанным в команде) целевым адресом команды `MUL` является комбинация регистров DX : AX.

В строке 30 результат прибавляется к локальной переменной, расположенной в стеке по адресу `-2(BP)`. Так как команда `MUL` не выполняет автоматическое

приращение своего операнда, это действие явно выполняется в строке 31. После этого регистр DI указывает на следующий элемент vec2.

Текущий этап работы программы завершается командой LOOP. Относительно значения регистра CX выполняется отрицательное приращение, и если после этого оно остается положительным, программа переходит к локальной метке 1 в строке 28. Локальная метка 1b означает поиск ближайшей метки 1 в обратном направлении от текущей позиции. После завершения цикла подпрограмма выталкивает возвращаемое значение из стека в регистр AX (строка 33), восстанавливает значение BP (строка 34) и возвращается к вызывающей программе (строка 35).

После вызова исполнение основной программы возобновляется с помощью команды MOV (переход к строке 12). Эта команда открывает последовательность из пяти команд, направленную на вывод результата. Системный вызов printf построен по модели функции printf стандартной библиотеки программирования языка C. В строках 13–15 в стек помещаются 3 аргумента: целочисленное значение, которое предполагается вывести, адрес форматной строки (pfmt) и код функции printf (127). Форматная строка pfmt содержит символ %d, указывающий на то, что целочисленная переменная, необходимая для форматирования, является аргументом вызова printf.

В строке 17 стек очищается. Так как начало программы находится в строке 5, где указатель стека был сохранен в регистре указателя базы, для очистки стека с тем же успехом можно запустить команду

```
MOV SP,BP
```

Преимущество такого решения состоит в том, что программисту не приходится следить за стеком. Применительно к основной программе это не слишком важно, но при работе с подпрограммами этот подход позволяет избавляться от ненужных данных, таких как устаревшие локальные переменные.

Подпрограмму vecmul можно включать в другие программы. Если имя исходного файла vecprod.s поместить в командой строке перед именем другого исходного файла на языке ассемблера, последний сможет обращаться к подпрограмме умножения двух векторов фиксированной длины. Во избежание дублирования предварительно рекомендуется исключить определения констант _EXIT и _PRINTF. Если заголовочный файл syscalnr.h подсоединен, писать определения констант системных вызовов в других местах нет необходимости.

Отладка программы вывода массива

Рассмотренные программы были простыми, но не содержали ошибок. В следующем примере мы покажем, как трассер помогает в отладке программ с ошибками. Наша следующая программа предназначена для вывода целочисленного массива, определенного после метки vec1. В ее первоначальной версии есть 3 ошибки. Для их выявления мы привлечем ассемблер и трассер, но сначала обсудим код.

Так как системные вызовы, а значит, и константы, с помощью которых эти вызовы можно различать по номерам, нужны любой программе, мы выделили определения констант с этими номерами в отдельный заголовочный файл, который включается в код в первой строке:

```
#include "../syscalnr.h"
```

Помимо прочего, в этом файле определены константы для следующих дескрипторов файлов:

```
STDIN = 0
STDOUT = 1
STDERR = 2
```

Они открываются в начале процесса, а в заголовке содержатся метки, указывающие на секции текста и данных. Этот файл имеет смысл включать в заголовок всех исходных ассемблерных файлов, поскольку имеющиеся в нем определения весьма востребованы. Если исходный код рассредоточен по нескольким файлам, ассемблер включает в него только одну версию заголовочного файла, за счет чего ситуации многократного определения констант удается избежать.

Программа *arrayprt* приведена в листинге В.4. Код не прокомментирован, так как мы предполагаем, что к настоящему моменту читатель уже в достаточной степени знаком с набором команд. В строке 4 адрес пустого стека помещается в регистр указателя базы — так предусматривается возможность очистки стека путем копирования указателя базы в указатель стека, что и выполняется в строке 10. В предыдущем примере (в строках 5–9) мы уже рассматривали ситуацию вычисления и введения в стек аргументов перед вызовом. В строках 22–25 регистры загружаются в подпрограмму.

Листинг В.4. Программа arrayprt перед отладкой

```
#include "../syscalnr.h"        ! 1

.SECT .TEXT                     ! 2
vecpstrt:                       ! 3
    MOV BP,SP                   ! 4
    PUSH vec1                   ! 5
    MOV CX,frmatstr-vec1        ! 6
    SHR CX                      ! 7
    PUSH CX                     ! 8
    CALL vecprint               ! 9
    MOV SP,BP                   ! 10
    PUSH0                       ! 11
    PUSH _EXIT                  ! 12
    SYS                         ! 13

.SECT .DATA                     ! 14
vec1: .WORD 3,4,7,11,3          ! 15
frmatstr: .ASCIZ "%s"           ! 16

frmatkop:                       ! 17
.ASCIZ "The array contains "    ! 18
frmatint: .ASCIZ " %d"          ! 19

.SECT .TEXT                     ! 20
vecprint:                       ! 21
    PUSH BP                     ! 22
    MOV BP,SP                   ! 23
    MOV CX,4(BP)                ! 24
    MOV BX,6(BP)                ! 25
    MOV SI,0                    ! 26
    PUSH frmatkop               ! 27
```

```
    PUSH frmatstr               ! 28
    PUSH _PRINTF                ! 29
    SYS                         ! 30
    MOV -4(BP),frmatint         ! 31
 1:  MOV DI,(BX)(SI)            ! 32
    MOV -2(BP),DI               ! 33
    SYS                         ! 34
    INC SI                      ! 35
    LOOP 1b                     ! 36
    PUSH '\n'                   ! 37
    PUSH _PUTCHAR               ! 38
    SYS                         ! 39
    MOV SP,BP                   ! 40
    RET                         ! 41
```

В строках 27–30 кода показано, как вывести символьную строку, а в строках 31–34 системный вызов `printf` выполнятся применительно уже к целочисленному значению. Адрес символьной строки вводится в стек в строке 27, а в строке 33 в стек вводится целочисленное значение. В обоих случаях адресом форматной строки выступает первый аргумент команды `PRINTF`. В строках 37–39 отдельный символ выводится при помощи системного вызова `putchar`.

Теперь попробуем ассемблировать и запустить программу. Для этого введем команду

```
as88 arrayprt.s
```

В результате появляется сообщение об ошибке операнда в строке 28 файла arrayprt.$. Этот файл генерируется ассемблером путем объединения включаемых файлов с исходным файлом; именно результирующий файл обрабатывается ассемблером. В сообщении об ошибке имеется в виду строка 28 именно этого объединенного файла. Изучение строки 28 файла arrayprt.s ничего не даст — нумерация строк в двух файлах не совпадает из-за включения в файл arrayprt.$ строк заголовочного файла. Строка 28 файла arrayprt.$ соответствует строке 7 arrayprt.s, так как включаемый заголовочный файл syscalnr.h содержит 21 строку.

В UNIX для поиска строки 28 в файле arrayprt.$ достаточно ввести команду

```
head -28 arrayprt.$
```

Эта команда выводит первые 28 строк объединенного файла. Соответственно, ошибку нужно искать в нижней строке листинга. Аналогичного эффекта можно добиться, просмотрев объединенный файл в текстовом редакторе. Таким образом, мы локализуем ошибку в строке 7 исходной программы, которая содержит команду `SHR`. Путем изучения табл. В.2 проблема легко обнаруживается: мы забыли указать величину смещения. Строка 7 после исправления должна выглядеть следующим образом:

```
SHR CX,1
```

Важно отметить, что ошибку нужно исправлять в файле arrayprt.s, а не в объединенном файле arrayprt.$, так как последний автоматически обновляется при каждом запуске ассемблера.

Следующая попытка ассемблировать исходный код, по идее, должна пройти успешно. Затем запускаем трассер командой

```
t88 arrayprt
```

В ходе трассировки замечаем, что выходные данные не согласуются с вектором, находящимся в сегменте данных. Вектор содержит значения 3, 4, 7, 11 и 3, в то время как на выходе последовательность начинается с 3, 1024,... Очевидно, что-то не так.

Чтобы найти ошибку, трассер можно запустить заново и шаг за шагом отслеживать состояние машины вплоть до появления неверного значения. Значения, которое требуется вывести, хранятся в памяти в строках 32 и 33. Строка вывода неверного значения — весьма удачное место для начала поисков. При втором проходе цикла становится заметно, что численное значение SI является нечетным, чего не должно быть по определению, так как индексирование производится по словам, а не по байтам. Таким образом, проблема локализуется в строке 35. Значение SI в ней приращивается на единицу, в то время как правильный шаг приращения — 2. Чтобы исправить ошибку, строку нужно изменить следующим образом:

```
ADD SI,2
```

После исправления выводимый список чисел не вызывает нареканий.

Тем не менее нас поджидает еще одна ошибка. После завершения вызова `vecprint` и возврата значения трассер отмечает ошибку в указателе стека. Очевидное решение — проверить, совпадает ли значение, вводимое в стек при вызове `vecprint`, со значением, находящимся на вершине стека при исполнении команды `RET` в строке 41. Как выясняется, они не совпадают. Таким образом, строку 40 следует заменить двумя новыми строками:

```
ADD SP,10
POP BP
```

Первая команда удаляет 5 слов, помещенных в стек в ходе вызова `vecprint`; таким образом открывается доступ к значению BP, сохраненному в строке 22. Путем выталкивания этого значения из стека мы восстанавливаем значение регистра BP, имевшее место перед вызовом, и получаем правильный адрес возврата. Теперь программа завершается корректно. Не секрет, что отладка кода на языке ассемблера — скорее искусство, чем наука, однако не стоит пренебрегать помощью трассера, который значительно упрощает процесс.

Обработка символьных строк и строковые команды

Основное назначение этого подраздела — показать механизм обработки повторяющихся строковых команд. В листингах В.5 и В.6 представлены простые программы для обработки символьных строк, strngcpy.s и reverspr.s, расположенные в папке examples.

Листинг В.5. Копирование строки (strngcpy.s)

```
.SECT .TEXT
stcstart:                       ! 1
    PUSH mesg1                  ! 2
    PUSH mesg2                  ! 3
    CALL strngcpy               ! 4
    ADD SP,4                    ! 5
    PUSH 0                      ! 6
    PUSH 1                      ! 7
    SYS                         ! 8
```

```
strngcpy:                       ! 9
    PUSH CX                     ! 10
    PUSH SI                     ! 11
    PUSH DI                     ! 12
    PUSH BP                     ! 13
    MOV BP,SP                   ! 14
    MOV AX,0                    ! 15
    MOV DI,10(BP)               ! 16
    MOV CX,- 1                  ! 17
    REPNZ SCASB                 ! 18
    NEG CX                      ! 19
    DEC CX                      ! 20
    MOV SI,10(BP)               ! 21
    MOV DI,12(BP)               ! 22
    PUSH DI                     ! 23
    REP MOVSB                   ! 24
    CALL stringpr               ! 25
    MOV SP,BP                   ! 26
    POP BP                      ! 27
    POP DI                      ! 28
    POP SI                      ! 29
    POP CX                      ! 30
    RET                         ! 31
.SECT .DATA                     ! 32
mesg1: .ASCIZ "Have a look\n"   ! 33
mesg2: .ASCIZ "qrst\n"          ! 34
.SECT .BSS
```

Листинг B.6. Вывод символьных строк задом наперед (reverspr.s)

```
#include "../syscalnr.h"        ! 1
    start: MOV DI,str           ! 2
    PUSH AX                     ! 3
    MOV BP,SP                   ! 4
    PUSH _PUTCHAR               ! 5
    MOVB AL,'\n'                ! 6
    MOV CX,-1                   ! 7
    REPNZ SCASB                 ! 8
    NEG CX                      ! 9
    STD                         ! 10
    DEC CX                      ! 11
    SUB DI,2                    ! 12
MOV SI,DI                       ! 13
    1: LODSB                    ! 14
    MOV (BP),AX                 ! 15
    SYS                         ! 16
    LOOP 1b                     ! 17
    MOVB (BP),'\n'              ! 18
    SYS                         ! 19
    PUSH 0                      ! 20
    PUSH _EXIT                  ! 21
SYS                             ! 22
.SECT .DATA                     ! 23
str: .ASCIZ "reverse\n"         ! 24
```

В листинге B.5 представлена программа копирования строки. Она вызывает подпрограмму `stringpr`, которую также можно найти в отдельном файле stringpr.s (листинга с этим файлом в нашем приложении нет). Чтобы ассемблировать программу, содержащую подпрограммы в отдельных файлах, в команде `as88` все исходные файлы следует указать, начиная с основной программы, которая определяет имена исполняемого и вспомогательных файлов. К примеру, чтобы ассемблировать программу, представленную в листинге B.5, нужно ввести команду

```
as88 strngcpy.s stringpr.s
```

Программа из листинга B.6 выводит символьные строки с обратным порядком следования символов. Мы рассмотрим обе программы по очереди.

В листинге B.5 строки кода пронумерованы, начиная с первой метки. Основная программа (строки 2–8) начинается с вызова подпрограммы `strngcpy` с двумя аргументами: исходной строкой `mesg2` и целевой строкой `mesg1`; подпрограмма копирует содержимое первой во вторую.

Теперь рассмотрим непосредственно подпрограмму `strngcpy`, которая начинается в строке 9 кода. Она рассчитана на то, что адреса целевого буфера и исходной строки вводятся в стек непосредственно перед ее вызовом. В строках 10–13 задействованные регистры сохраняются путем передачи их значений в стек; впоследствии, в строках 27–30, их предполагается восстановить. В строке 14 значения SP и BP копируются стандартным методом. После этого в BP можно загружать аргументы. В строке 26 стек очищается уже знакомым нам способом — значение BP копируется в SP.

Центральным местом подпрограммы является команда `REP MOVSB`, расположенная в строке 24 кода. Команда `MOVSB` перемещает байт, на который указывает регистр SI, по адресу памяти, определяемому регистром DI. После этого содержимое обоих регистров увеличивается на единицу. Команда `REP` формирует цикл, в котором исполняется команда `MOVSB`; после перемещения каждого байта она уменьшает значение CX на 1. Цикл завершается при достижении CX нулевого значения.

Перед исполнением `REP MOVSB` необходимо подготовить регистры, что и делается в строках 15–22. Индекс источника, SI, копируется из аргумента в стек в строке 21; индекс приемника, DI, определяется в строке 22. Получить значение CX несколько сложнее. Следует иметь в виду, что конец символьной строки обозначается нулевым байтом. Команда `MOVSB` не влияет на состояние нулевого флага, в отличие от команды `SCASB` (просмотр байтовой строки). Последняя сравнивает значение, на которое указывает DI, со значением AL и выполняет приращение DI «на лету». Подобно `MOVSB`, она относится к числу повторяющихся команд. В строке 15 очищаются регистры AX и AL, в строке 16 из стека выбирается указатель на DI, а в строке 17 регистр CX инициализируется значением –1. В строке 18 находится команда `REPNZ SCASB`, выполняющая операцию сравнения в контексте цикла и в случае равенства устанавливающая нулевой флаг. При каждом проходе цикла выполняется отрицательное приращение CX, а при установлении нулевого флага цикл останавливается — команда `REPNZ` проверяет и нулевой флаг, и CX. Количество проходов цикла с командой `MOVSB`, таким образом, определяется как разность текущего значения CX и предыдущего значения –1 (строки 19–20).

Использование двух повторяющихся команд выглядит не слишком изящно, но такова цена решения, основанного на принципе независимости кодов условий от команд перемещения. При исполнении циклов проводится приращение индексных регистров, и для этих целей флаг направления должен быть сброшен.

В строках 23 и 25 кода скопированная символьная строка выводится при помощи подпрограммы `stringpr`, имеющейся в папке examples. Она достаточно проста, поэтому обсуждать ее здесь мы не будем.

В программе вывода символьных строк в обратном порядке, показанной в листинге B.6, строка 1 кода содержит все стандартные номера системных вызовов. В строке 3 в стек помещается фиктивное значение, а в строке 4 указатель базы (BP) начинает указывать на текущую вершину стека. Эта программа выводит ASCII-символы по одному, а потому в стек вводится численное значение `_PUTCHAR`. Обратите внимание: BP указывает на символ, который предполагается отобразить во время вызова `SYS`.

В строках 2, 6 и 7 регистры DI, AL и CX подготавливаются к исполнению повторяющейся команды `SCASB`. Регистр счетчика и целевой индекс загружаются так же, как это происходит в программе копирования строки, за тем лишь исключением, что в регистр AL вместо нуля помещается символ новой строки. Таким образом, команда `SCASB` сравнивает значения символов строки `str` с `\n`, а не с нулем, и в случае соответствия устанавливает нулевой бит.

Команда `REP SCASB` выполняет приращение регистра DI, так что после совпадения целевой индекс указывает на символ нуля, следующий за символом новой строки. В строке 12 выполняется отрицательное приращение DI на 2; в результате этот регистр указывает на последнюю букву слова.

Если сканирование символьной строки идет в обратном порядке, а отображается она посимвольно, значит, наша задача решена; в таком случае флаг направления в строке 10 переустанавливается, и начинается обратная регулировка индексных регистров в строковых командах. Теперь команда `LODSB` в строке кода 14 копирует символ в AL, а в строке 15 этот символ помещается в стек рядом с `_PUTCHAR`, что позволяет команде `SYS` отобразить его.

Команды, находящиеся в строках 18 и 19, выводят новую строку, и программа традиционно закрывается вызовом `_EXIT`.

В текущей версии программы, впрочем, содержится ошибка. Ее можно найти путем пошаговой трассировки.

Команда `/str` помещает строку `str` в поле данных трассера. Так как числовое значение адреса данных известно, мы можем выяснить, как меняются значения в индексных регистрах в отношении положения символьной строки.

Ошибка, впрочем, обнаруживается только после многократного нажатия клавиши возврата каретки. Команды трассера помогают сократить время локализации ошибки. Запустите программу трассировки и введите команду `13`, которая переместит нас в середину цикла. Далее с помощью команды `b` установим контрольную точку в строке 15. Создав две новые строки, мы увидим, что в поле вывода отображается последняя буква «e». Запустив команду `r`, мы заставим программу трассировки работать вплоть до следующей контрольной точки или до завершения процесса. Таким образом, последовательно запуская команду `r`, можно видеть все буквы, пока мы не подберемся к решению проблемы. С этого

момента программу трассировки можно будет перевести в пошаговый режим и проанализировать происходящее при исполнении важнейших команд.

Мы также можем установить дополнительную контрольную точку в той или иной строке кода, но при этом нужно учитывать включение файла syscalnr.h, приводящее к смещению номеров строк кода на 20. Следовательно, чтобы установить контрольную точку, скажем, в строке 16, нужно ввести команду `36b`. Впрочем, от этого неуклюжего способа лучше отказаться в пользу установки глобальной метки `start` в строке 2 перед расположенной в ней командой — тогда контрольную точку в нужной строке кода можно будет установить командой

```
/start + 14b
```

К тому же такой способ установки контрольной точки позволяет не обращать внимания на размер включаемого файла.

Таблицы диспетчеризации

В некоторых языках программирования существуют операторы выбора, позволяющие выбирать один из нескольких возможных вариантов действий в зависимости от численного значения переменной. Подобного рода многовариантное ветвление иногда оказывается полезным и в программах на языке ассемблера. Возьмем, к примеру, набор подпрограмм системных вызовов, объединенных в программе перехвата исключений SYS. Пример программирования многовариантного ветвления на языке ассемблера 8088 показан в коде программы jumptbl.s, представленном в листинге В.7.

Листинг В.7. Реализация многовариантного ветвления при помощи таблицы диспетчеризации

```
#include "../syscalnr.h"                                    ! 1
.SECT .TEXT                                                 ! 2
jumpstrt:                                                   ! 3
    PUSH strt                                               ! 4
    MOV BP,SP                                               ! 5
    PUSH _PRINTF                                            ! 6
    SYS                                                     ! 7
    PUSH _GETCHAR                                           ! 8
1:  SYS                                                     ! 9
    CMP AX,5                                                ! 10
    JL 8f                                                   ! 11
    CMPB AL,'0'                                             ! 12
    JL 1b                                                   ! 13
    CMPB AL,'9'                                             ! 14
    JLE 2f                                                  ! 15
    MOVB AL,'9'+1                                           ! 16
2:  MOV BX,AX                                               ! 17
    AND BX,0Xf                                              ! 18
    SAL BX,1                                                ! 19
    CALL tbl(BX)                                            ! 20
    JMP 1b                                                  ! 21
8:  PUSH 0                                                  ! 22
    PUSH _EXIT                                              ! 23
    SYS                                                     ! 24
rout0: MOV AX,mes0                                          ! 25
```

```
      JMP 9f                                                    ! 26
rout1: MOV AX,mes1                                              ! 27
      JMP 9f                                                    ! 28
rout2: MOV AX,mes2                                              ! 29
      JMP 9f                                                    ! 30
rout3: MOV AX,mes3                                              ! 31
      JMP 9f                                                    ! 32
rout4: MOV AX,mes4                                              ! 33
      JMP 9f                                                    ! 34
rout5: MOV AX,mes5                                              ! 35
      JMP 9f                                                    ! 36
rout6: MOV AX,mes6                                              ! 37
      JMP 9f                                                    ! 38
rout7: MOV AX,mes7                                              ! 39
      JMP 9f                                                    ! 40
rout8: MOV AX,mes8                                              ! 41
      JMP 9f                                                    ! 42
erout: MOV AX,emes                                              ! 43
9:    PUSH AX                                                   ! 44
      PUSH _PRINTF                                              ! 45
      SYS                                                       ! 46
      ADD SP,4                                                  ! 47
      RET                                                       ! 48

.SECT .DATA                                                     ! 49
tbl: .WORD rout0,rout1,rout2,rout3,rout4,rout5,
           rout6,rout7,rout8,rout8,erout                        ! 50
mes0: .ASCIZ "This is a zero.\n"                                ! 51
mes1: .ASCIZ "How about a one.\n"                               ! 52
mes2: .ASCIZ "You asked for a two.\n"                           ! 53
mes3: .ASCIZ "The digit was a three.\n"                         ! 54
mes4: .ASCIZ "You typed a four.\n"                              ! 55
mes5: .ASCIZ "You preferred a five.\n"                          ! 56
mes6: .ASCIZ "A six was encountered.\n"                         ! 57
mes7: .ASCIZ "This is number seven.\n"                          ! 58
mes8: .ASCIZ "This digit is not accepted as an octal.\n"        ! 59
emes: .ASCIZ "This is not a digit. Try again.\n"                ! 60
strt: .ASCIZ "Type an octal digit with a return. Stop on end of file.\n"! 61
```

Программа начинается с вывода символьной строки с меткой `strt`, предлагающей пользователю ввести восьмеричную цифру (строки 4–7 кода). Затем из файла стандартного ввода считывается символ (строки 8 и 9). Если значение AX оказывается меньше 5, программа интерпретирует это как маркер конца файла, переходит к метке 8 в строке 22 и завершается с кодом состояния 0.

Если введенный символ не является маркером конца файла, исследуется введенный символ в регистре AL. Любой символ, меньший цифры 0, считается разделителем и при переходе в строке кода 13 игнорируется; после чего извлекается следующий символ. Любой символ, больший цифры 9, считается неверным. В строке 16 он преобразуется в ASCII-символ двоеточия, который в последовательности ASCII-символов идет сразу после цифры 9.

Таким образом, в строке 17 в регистре AX находится значение в диапазоне между цифрой 0 и двоеточием. Это значение копируется в регистр BX. В строке 18 команда `AND` маскирует все биты, кроме четырех младших, в результате

остается число между 0 и 10 (это связано с тем, что нулю соответствует ASCII-код 0x30). Так как в таблице мы собираемся провести индексирование слов, а не байтов, значение BX умножается на 2 путем сдвига влево в строке 19.

В строке 20 находится команда вызова. Действительный адрес определяется прибавлением значения BX к численному значению метки tbl, а содержимое этого объединенного адреса загружается в счетчик команд PC.

Программа выбирает одну из десяти подпрограмм в зависимости от символа, извлекаемого из стандартного ввода. Каждая из этих подпрограмм размещает в стеке адрес того или иного сообщения, а затем переходит к общему для всех вызову системной подпрограммы _PRINTF.

Чтобы разобраться в происходящем, следует иметь в виду, что команды JMP и CALL загружают в регистр PC некий адрес текстового сегмента. Этот адрес представляет собой двоичное число, а в ходе ассемблирования все адреса заменяются соответствующими двоичными значениями. Двоичные значения, в свою очередь, помогают инициализировать массив в сегменте данных, что и делается в строке 50. Таким образом, массив, начинающийся с метки tbl, содержит начальные адреса подпрограмм rout0, rout1, rout2 и т. д., по два байта в каждом. Наличие 2-байтовых адресов объясняет необходимость в сдвиге на 1 бит, произведенном в строке 19. Таблицы такого типа часто называют **таблицами диспетчеризации**.

Механизм работы таких программ демонстрирует подпрограмма erout (строки 43–48 кода). Она обрабатывает числа, выходящие за пределы допустимого диапазона. Во-первых, в строке 43 в стек помещается адрес сообщения (в AX). Затем в стек отправляется число системного вызова _PRINTF. Далее выполняется системный вызов, стек очищается, и программа выполняет возврат. Остальные десять подпрограмм — от rout0 до rout8 — загружают адреса своих сообщений в регистр AX, а затем переходят ко второй строке erout, выводят сообщения и завершают подпрограмму.

Один из способов адаптировать программу к таблицам диспетчеризации — изменить ее таким образом, чтобы из вводимых символов формировалось осмысленное сообщение. К примеру, для всех символов, кроме восьмеричных цифр, может выдаваться сообщение об ошибке.

Буферизованный и произвольный доступ к файлам

Программа InFilBuf.s, представленная в листинге В.8, являет собой пример произвольного файлового ввода-вывода. Она допускает, что файл состоит из произвольного числа символьных строк, каждая из которых потенциально может отличаться по длине от других. Сначала эта программа считывает файл и формирует таблицу, в которой запись n отражает положение начала строки n в файле. Впоследствии можно запросить конкретную символьную строку, отыскать запись для нее в таблице и считать с помощью системных вызовов lseek и read. Имя файла при стандартном вводе указывается в первой вводимой строке. Программа состоит из нескольких относительно независимых кодовых фрагментов, которые можно адаптировать для иных целей.

Листинг В.8. Программа, реализующая буферизованный и произвольный доступ к файлу

```
#include "../syscalnr.h"       ! 1
bufsiz = 512                   ! 2
.SECT .TEXT                    ! 3
infbufst:                      ! 4
    MOV BP,SP                  ! 5
    MOV DI,linein              ! 6
    PUSH _GETCHAR              ! 7
1:  SYS                        ! 8
    CMPB AL,'\n'               ! 9
    JL 9f                      ! 10
    JE 1f                      ! 11
    STOSB                      ! 12
    JMP 1b                     ! 13
1:  PUSH 0                     ! 14
    PUSH linein                ! 15
    PUSH _OPEN                 ! 16
    SYS                        ! 17
    CMP AX,0                   ! 18
    JL 9f                      ! 19
    MOV (fildes),AX            ! 20
    MOV SI,linh+2              ! 21
    MOV BX,0                   ! 22
1:  CALL fillbuf               ! 23
    CMP CX,0                   ! 24
    JLE 3f                     ! 25
2:  MOVB AL,'\n'               ! 26
    REPNE SCASB                ! 27
    JNE 1b                     ! 28
    INC (count)                ! 29
    MOV AX,BX                  ! 30
    SUB AX,CX                  ! 31
    XCHG SI,DI                 ! 32
    STOS                       ! 33
    XCHG SI,DI                 ! 34
    CMP CX,0                   ! 35
    JNE 2b                     ! 36
    JMP 1b                     ! 37
9:  MOV SP,BP                  ! 38
    PUSH linein                ! 39
    PUSH errmess               ! 40
    PUSH _PRINTF               ! 41
    SYS                        ! 42
    PUSH _EXIT                 ! 43
    PUSH _EXIT                 ! 44
    SYS                        ! 45
3:  CALL getnum                ! 46
    CMP AX,0                   ! 47
    JLE 8f                     ! 48
    MOV BX,(curlin)            ! 49
    CMP BX,0                   ! 50
    JLE 7f                     ! 51
    CMP BX,(count)             ! 52
    JG 7f                      ! 53
```

продолжение ⇩

Листинг В.8 *(продолжение)*

```
    SHL BX,1                    ! 54
    MOV AX,linh-2(BX)           ! 55
    MOV CX,linh(BX)             ! 56
    PUSH 0                      ! 57
    PUSH 0                      ! 58
    PUSH AX                     ! 59
    PUSH (fildes)               ! 60
    PUSH _LSEEK                 ! 61
    SYS                         ! 62
    SUB CX,AX                   ! 63
    PUSH CX                     ! 64
    PUSH buf                    ! 65
    PUSH (fildes)               ! 66
    PUSH _READ                  ! 67
    SYS                         ! 68
    ADD SP,4                    ! 69
    PUSH 1                      ! 70
    PUSH _WRITE                 ! 71
    SYS                         ! 72
    ADD SP,14                   ! 73
    JMP 3b                      ! 74
8:  PUSH scanerr                ! 75
    PUSH _PRINTF                ! 76
    SYS                         ! 77
    ADD SP,4                    ! 78
    JMP 3b                      ! 79
7:  PUSH 0                      ! 80
    PUSH _EXIT                  ! 81
    SYS                         ! 82
fillbuf:                        ! 83
    PUSH bufsiz                 ! 84
    PUSH buf                    ! 85
    PUSH (fildes)               ! 86
    PUSH _READ                  ! 87
    SYS                         ! 88
    ADD SP,8                    ! 89
    MOV CX,AX                   ! 90
    ADD BX,CX                   ! 91
    MOV DI,buf                  ! 92
    RET                         ! 93

getnum:                         ! 94
    MOV DI,linein               ! 95
    PUSH _GETCHAR               ! 96
1:  SYS                         ! 97
    CMPB AL,'\n'                ! 98
    JL 9b                       ! 99
    JE 1f                       ! 100
    STOSB                       ! 101
    JMP 1b                      ! 102
1:  MOVB (DI),' '               ! 103
    PUSH curlin                 ! 104
    PUSH numfmt                 ! 105
    PUSH linein                 ! 106
```

```
    PUSH _SSCANF              ! 107
    SYS                       ! 108
    ADD SP,10                 ! 109
    RET                       ! 110

.SECT .DATA                   ! 111
errmess:                      ! 112
.ASCIZ "Open %s failed\n"     ! 113
numfmt: .ASCIZ "%d"           ! 114
scanerr:                      ! 115
.ASCIZ "Type a number.\n"     ! 116
.ALIGN2                       ! 117
.SECT .BSS                    ! 118
linein:  .SPACE 80            ! 119
fildes:  .SPACE2              ! 120
linh:  .SPACE 8192            ! 121
curlin:  .SPACE4              ! 122
buf:    .SPACE bufsiz+2       ! 123
count:  .SPACE2               ! 124
```

В первых пяти строках кода определяются номера системных вызовов и размер буфера, а указатель базы, как обычно, настраивается на вершину стека. В строках 6–13 из стандартного ввода считывается имя файла, а затем оно сохраняется в символьной строке с меткой `linein`. Если имя файла не закрыто новой строкой, выводится сообщение об ошибке, а процесс заканчивается с ненулевым состоянием. Все эти действия отражены в строках 38–45. Обратите внимание, что адрес имени файла помещается в стек в строке 39, адрес сообщения об ошибке — в строке 40. Само сообщение об ошибке (представленное в строке 113) представляет собой запрос на строку `$s` в формате `_PRINTF`. Здесь же выполняется вставка содержимого строки `linein`.

В случае успешного копирования имени файла его открытие производится в строках 14–20. Если вызов `open` завершается с ошибкой, возвращаемое значение становится отрицательным, и для отображения сообщения об ошибке производится переход к метке 9 в строке 28. Если вызов проходит успешно, возвращаемым значением является дескриптор файла, который сохраняется в переменной `fildes`. Этот дескриптор понадобится при последующем выполнении вызовов `read` и `lseek`.

Далее файл считывается блоками по 512 байт, каждый из которых сохраняется в буфере `buf`. На самом деле, под буфер выделяется на два байта больше необходимого объема (512 байт), но сделано это лишь для того, чтобы показать способ размещения символической константы и целого числа в одном выражении (строка 123). Аналогичным образом, в строке 21 в регистр SI загружается адрес следующего элемента массива `linh`, в результате на дне массива остается машинное слово с нулевым значением. Регистр BX получает адрес файла первого непрочитанного символа файла, а, значит, в строке 22 перед первым наполнением буфера он инициализируется нулем.

За наполнение буфера отвечает подпрограмма `fillbuf`, размещенная в строках 83–93. После введения в стек аргументов `read` делается запрос на системный вызов, который помещает число фактически считанных символов в регистр AX. Это число копируется в CX, и впоследствии по значению этого регистра можно

будет узнать число оставшихся в буфере символов. Положение в файле первого непрочитанного символа хранится в регистре BX, и в строке 91 значение CX прибавляется к значению BX. В строке 92 дно буфера помещается в DI; таким образом, осуществляется подготовка к просмотру буфера на предмет следующего символа новой строки.

После возврата из `fillbuf` в строке 24 проводится проверка на то, были ли какие-либо символы фактически считаны. При отрицательном ответе происходит переход из цикла чтения с буферизацией ко второй части программы в строке 25.

После этого начинается просмотр буфера. Символ `\n` загружается в регистр AL в строке 26, а уже в следующей строке это значение просматривается в цикле `REP SCASB` и сравнивается с буферизованными символами. Выход из цикла может произойти в двух случаях: когда в регистре CX оказывается нулевое значение или когда просматриваемый символ оказывается символом новой строки. При установленном нулевом флаге последним просмотренным символом оказывается `\n`, а положения в файле текущего символа (расположенного после перевода строки) сохраняется в массиве `linh`. Далее происходит приращение счетчика, а положение в файле определяется по значению BX и числу символов, оставшихся в CX (строки 29–31 кода). В строках 32–34 выполняется сохранение, но так как для команды `STOS` целевым адресом является не регистр SI, а регистр DI, перед и после вызова команды `STOS` эти регистры меняются местами. В строках 35–37 производится проверка оставшихся в буфере данных, после чего в зависимости от значения CX выполняется переход.

По достижении конца файла в нашем распоряжении оказывается полный список положений в файле начальных элементов строк. Так как массив `linh` начинается с нулевого слова, нам известно, что первая строка начинается с адреса 0, следующая строка находится в положении `linh` + 2 и т. д. Размер строки `n` можно вычислить путем вычитания начального адреса строки `n` из начального адреса строки `n` + 1.

В оставшейся части программы считывается номер символьной строки, эта строка передается в буфер, после чего она выводится при помощи вызова `write`. Все необходимые для выполнения этих операций данные имеются в массиве `linh`, где каждая `n`-ная запись содержит начальное положение строки `n` в файле. Если номер запрошенной символьной строки равен нулю или выходит за пределы допустимого диапазона, программа завершается путем перехода к метке 7.

Эта часть программы начинается с вызова подпрограммы `getnum` (строка 46 кода). Она считывает символьную строку из стандартного ввода и сохраняет в буфере `linein` (строки 95–103 кода). Далее подготавливается вызов `SSCANF`. Принимая во внимание обратный порядок следования аргументов, в стек сначала помещается адрес буфера `curlin`, где можно разместить целочисленное значение, затем адрес форматной строки для целочисленного представления `numfmt` и, наконец, адрес буфера `linein`, в котором содержится число в десятичном представлении. Если это возможно, системная подпрограмма `SSCANF` помещает двоичное значение в `curlin`. В случае ошибки она возвращает нулевое значение AX. Проверка возвращаемого значения проводится в строке 48; при ошибке программа генерирует сообщение об ошибке посредством метки 8.

Если подпрограмма `getnum` возвращает действительное целочисленное значение в `curlin`, оно сначала копируется в BX. Далее это значение проверяется на

предмет принадлежности к допустимому диапазону (строки 49–53 кода). Если обнаруживается, что номер строки выходит за пределы этого диапазона, происходит выход (EXIT).

Далее необходимо определить конечное положение данной строки в файле и число считываемых файлов; с этой целью значение BX умножается на 2 посредством сдвига влево (SHL). В строке 55 положение текущей символьной строки в файле копируется в регистр AX. Затем положение следующей символьной строки помещается в регистр CX и на его основе вычисляется число байтов в текущей строке.

Для произвольного чтения данных из файла применяется вызов lseek; он устанавливает смещение к байту, который предполагается прочесть в следующую очередь. Подпрограмма lseek исполняется относительно начала файла, и в этой связи в строке 57 в стек помещается нулевой аргумент. Следующим аргументом является смещение в файле. По определению этот аргумент является длинным (32-разрядным) целым, поэтому сначала в стек вводится нулевое слово, а затем — значение AX (строки 58 и 59 кода); таким образом формируется 32-разрядное целочисленное значение. Далее в строке 62 в стек отправляются дескриптор файла и код LSEEK, а также выполняется вызов. Возвращаемое значение LSEEK определяет текущее положение в файле, а найти его можно в комбинации регистров DX : AX. Если число умещается в рамки машинного слова (а размер файла не превышает 65 536 байт, по-другому быть и не может), адрес помещается в AX; следовательно, если вычесть значение этого регистра из CX (строка 63), мы получим число байтов, которые необходимо прочесть для помещения строки в буфер.

Все остальное очень просто. В строках 64–68 строка считывается из файла, а затем при помощи дескриптора 1 файла в строках 70–72 она записывается в файл стандартного вывода. Следует иметь в виду, что после частичной очистки стека, которая выполняется в строке 69, значения счетчика и буфера в нем остаются. Наконец, в строке 73, указатель стека сбрасывается полностью, после чего осуществляются обратный переход к метке 3 и очередной вызов getnum.

БЛАГОДАРНОСТИ

Ассемблер, к которому мы обращаемся в этом приложении, входит в набор инструментов Amsterdam Compiler Kit. Полную версию этого набора можно найти по адресу www.cs.vu.nl/ack. Мы признательны людям, принимавшим участие в его первоначальной разработке: Юхану Стивенсону (Johan Stevenson), Хансу Шамини (Hans Schaminee) и Хансу де Райсу (Hans de Vries). Мы в большом долгу перед Сирилем Якобсом (Ceriel Jacobs), который, занимаясь сопровождением этого программного пакета, несколько раз по нашей просьбе адаптировал его для применения в образовательном процессе, и Элтом Огстоном (Elth Ogston), читавшем рукопись и проверившим примеры и задания.

Кроме того, мы хотим поблагодарить Роберта ван Ренессе (Robbert van Renesse) и Яна-Марка Вамса (Ian-Mark Wams) — разработчиков трассеров для PDP-11 и Motorola 68000 соответственно. Многие высказанные ими идеи были учтены при разработке нашего трассера. Естественно, самых теплых слов заслуживают все наши ассистенты и операторы систем, в течение многих лет помогавшие нам в преподавании языков ассемблера.

Вопросы и задания

1. Каковы значения регистров AH и AL после исполнения команды `MOV AX, 702`?
2. Значение регистра CS — 4. Каков диапазон абсолютных адресов памяти в данном сегменте кода?
3. Назовите максимальный адрес памяти, по которому может обратиться 8088.
4. Предположим, что CS = 40, DS = 8000, а IP = 20.
   1) Каков абсолютный адрес следующей команды?
   2) Какое слово памяти загружается в регистр AX при исполнении команды `MOV AX, (2)`?
5. Подпрограмма с тремя целочисленными аргументами вызывается согласно описанной в этом приложении последовательности вызова — вызывающая программа помещает в стек аргументы в обратном порядке, а затем исполняет команду `CALL`. Затем вызываемая программа сохраняет прежнее значение BP и вводит новое значение, указывающее на прежнее. Далее в отношении указателя стека выполняется отрицательное приращение; тем самым выделяется пространство под размещение локальных переменных. Принимая во внимание эту последовательность, назовите команду, необходимую для перемещения первого аргумента в регистр AX.
6. На рис. B.1 в качестве операнда выступает выражение `de – hw`. Это значение выражает разность двух меток. Возможна ли ситуация, в которой действительным операндом могло бы стать выражение `de + hw`? Аргументируйте свой ответ.
7. Напишите на языке ассемблера код, позволяющий вычислить следующее выражение:
   $= a + b + 2$
8. Некая C-функция вызывается следующим образом:

   ```
   footbar(x,y);
   ```

   Напишите на языке ассемблера код, позволяющий выполнить этот вызов.
9. Напишите на языке ассемблера программу, которая на входе получает выражения, состоящие из целого числа, оператора и еще одного целого числа, а на выходе предоставляет значения этих выражений. Допускается применение операторов +, -, × и /.

Алфавитный указатель

L

M

N

O

P

R

S

T

U

Б

В

Г

Д

Е

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Э. Таненбаум, Т. Остин

Архитектура компьютера

6-е издание

Серия «Классика computer science»

Перевел с английского Е. Матвеев

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Е. Волошина*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Импортер в Беларусь: ООО «ПИТЕР М», РБ, 220020, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс 208 80 01.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Подписано в печать 24.11.20. Формат 70x100/16. Бумага писчая. Усл. п. л. 65,790. Доп. тираж. Заказ
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87