СЕРИЯ

COMPUTER ORGANIZATION

Fifth Edition

Carl Hamacher, Zvonko Vranesic,
Safwat Zaky

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

КЛАССИКА COMPUTER SCIENCE

К. ХАМАХЕР, З. ВРАНЕШИЧ, С. ЗАКИ

ОРГАНИЗАЦИЯ ЭВМ

5-Е ИЗДАНИЕ

bhv® ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2003

ББК 32.973.2
УДК 681.3
Х18

Х18 **Организация ЭВМ. 5-е изд.** / К. Хамахер, З. Вранешич, С. Заки. — СПб.: Питер; Киев: Издательская группа BHV, 2003. — 848 с.: ил. — (Серия «Классика computer science»).

ISBN 5-8046-0162-8
ISBN 966-552-122-5

Очередное издание книги всемирно известных авторов отражает их богатый опыт преподавания курса «Организация ЭВМ» в Университете г. Торонто. В книге на конкретных примерах современных устройств подробно описана архитектура компьютеров и строение их компонентов: процессоров, блоков памяти, устройств ввода-вывода. Аппаратные средства рассматриваются с учетом их взаимодействия с системным программным обеспечением. Большое внимание уделено современным компьютерным технологиям, новым стандартам памяти, а также компьютерным системам на базе встроенных процессоров и мультипроцессорным системам параллельного действия. Отдельная глава посвящена машинным кодам и языку ассемблера.

Книга может быть рекомендована как учебное пособие для студентов, обучающихся по направлению «Информатика и вычислительная техника», а также всем, кто интересуется устройством современных компьютеров.

ББК 32.973.2
УДК 681.3

ISBN 0-07-232086-9 (англ.)
ISBN5-8046-0162-8
ISBN 966-552-122-5

Краткое содержание

Содержание

Об авторах

Карл Хамахер (Carl Hamacher) имеет степень бакалавра в области инженерной физики (получил в Университете штата Ватерлоо, Канада) и степень магистра в области электронной техники (получил в Сиракузском университете, штат Нью-Йорк). В течение ряда лет (1968–1990) он преподавал в Университете города Торонто, где занимал должность профессора факультета электротехники и компьютерных наук. С 1984 по 1988 год возглавлял Исследовательский институт компьютерных систем, а с 1988 по 1990 — там же отдел инженерных наук. В январе 1991 года Карл получил должность профессора и возглавил факультет прикладных наук в Королевском университете. С этого времени он стал заниматься проблемами электротехники и компьютерной инженерии. Карл постоянно сотрудничает с учеными из других известных университетов и лабораторий, в частности он принимал участие в разработках исследовательской лаборатории IBM в Сан-Хосе, Калифорния (1978–1979), лаборатории схемотехники, поддерживающей связи с Университетом города Гренобля, Франция (1986), Университета штата Калифорния в Риверсайде (1996–1997), а также лаборатории LIP6 Парижского университета.

Объектами исследовательских интересов Карла Хамахера являются мультипроцессорные и мультикомпьютерные системы, а также коммуникационные сети высокопроизводительных вычислительных систем.

Звонко Вранешич (Zvonko Vranesic) имеет степени бакалавра, магистра и доктора Университета города Торонто. С 1963 по 1965 год работал инженером-конструктором в Northern Electric Co., Ltd. в Брамалеа, штат Онтарио. Преподавать в университете он начал в 1968 году, где сначала занимал должность профессора факультетов электротехники и компьютерной инженерии, компьютерных наук, а затем возглавил там же факультет инженерных наук (1995–2000). Звонко постоянно поддерживает научные связи с коллегами из других высших учебных заведений, в том числе из Кембриджского и Парижского университетов. С 2000 года он, получив должность главного инженера в Altera Corporation, Торонто, занимается вопросами программного обеспечения.

К сфере нынешних исследовательских интересов Звонко Вранешича относятся компьютерная архитектура, технологии VLSI и многозначные логические системы. Он является соавтором таких популярных книг, как «Fundamentals of Digital Logic with VHDL Design», «Microcomputer Structures» и «Field-Programmable Gate Arrays». За новаторский вклад в лабораторное обучение студентов Звонко в 1990 году был отмечен специальной премией.

Сафват Заки (Safwat Zaky) получил степень бакалавра в области электроинженерии и в области математики в Университете города Карио, Египет, а также степени магистра и доктора в области электроинженерии в Университете города Торонто. С 1969 по 1972 год Сафват, будучи сотрудником научно-исследовательского центра в городе Брамалеа, занимался проблемами применения электронной оптики и магнетизма для хранения информации и коммутации телефонных соединений. С 1973 года преподает в Университете города Торонто, где сейчас занимает должность профессора на факультете электротехники и компьютерной инженерии, а также на факультете компьютерных наук. Кроме того, два года, с 1980 по 1981, он являлся главным инспектором компьютерной лаборатории Кембриджского университета.

К области научных интересов Сафвата относятся архитектура компьютера, надежность цифровых схем и электромагнитная совместимость. Он является соавтором книги «Microcomputer Structures», награжден медалью IEEE Third Millennium.

Предисловие

Эта книга создавалась как учебное пособие по архитектуре компьютеров и вычислительных систем. Студентам, для которых она предназначена в первую очередь, достаточно иметь базовые знания в области программирования на языках высокого уровня. Так как во многих учебных заведениях при изучении устройства компьютеров обычно проходят вводный курс по цифровым логическим схемам, эта тема в основной части издания не рассматривается. Но для тех, кто с ней не знаком, приводится довольно обстоятельное приложение.

Мы, авторы книги, имеем немалый опыт преподавания, в частности такого предмета, как устройство компьютеров, студентам, обучающимся по специальностям: электротехника, компьютерные науки и компьютерная инженерия, а также специализирующимся в области разработки программного обеспечения. К вопросам подбора и изложения материала мы всегда подходили с практической точки зрения. Работая над настоящим изданием, мы старались проиллюстрировать важнейшие аспекты устройства компьютеров примерами, взятыми из документации по реальным коммерческим компьютерным системам. Большая часть примеров иллюстрирует работу вычислительных систем на базе процессоров ARM, Motorola 680X0, Intel Pentium и Sun UltraSPARC.

Важно понимать, что процесс разработки компьютерных систем не укладывается в простую схему применения оптимальных алгоритмов. Многие инженерные решения основываются на эвристических предположениях и практическом опыте. При их принятии учитывается соотношение стоимости и производительности, с одной стороны, а также сложности аппаратного и программного обеспечения — с другой, причем диапазон альтернативных вариантов очень широк. Все эти аспекты и соображения мы хотим донести до читателя.

Подробно освещая многие вопросы, казалось бы даже очевидные, мы старались вызвать у студентов интерес к углубленному изучению темы и показать, как много может быть скрыто за простыми, на первый взгляд, положениями. Этой цели как раз и служат предлагаемые нами реальные, подробно документированные примеры. Одним из наиболее эффективных способов изучения структуры и организации компьютеров традиционно считается рассмотрение блок-схем. Однако если изложение материала будет основано только на общем анализе блок-схем, у читателя сложится чересчур упрощенное представление об обсуждаемых вопросах. Поэтому мы сопровождаем описание каждой блок-схемы подробным рассказом о нескольких альтернативных способах ее реализации.

Книга рассчитана на изучение предмета в течение одного семестра в сочетании с другими курсами технических и компьютерных наук. Она будет полезна для студентов, изучающих как аппаратную составляющую компьютера, так и программную. И хотя основное внимание уделяется в первую очередь аппаратному обеспечению, мы рассмотрим и многие вопросы программирования. В частности, речь пойдет об основах построения компиляторов и операционных систем, о проблемах, связанных с эффективностью выполнения команд, координированием параллельных операций на системном уровне и работой приложений, функционирующих в реальном масштабе времени. Любому специалисту необходимо понимать принципы взаимодействия аппаратного и программного обеспечения, знать критерии поиска оптимального баланса между двумя составляющими компьютерной системы.

Содержание книги

Ниже перечислены основные вопросы, затронутые в каждой из глав книги. Сначала (первые восемь глав) рассматриваются основные устройства и принципы функционирования компьютеров, а также методы и средства оценки их производительности. Четыре последние главы посвящены встраиваемым системам, периферийным устройствам, семействам процессоров и большим компьютерным системам.

В главе 1 студенты познакомятся с аппаратным и программным обеспечением компьютеров. Здесь вводятся те термины, которыми мы будем оперировать и которые будут определены более точно по ходу изложения соответствующей темы. В частности, в данной главе рассматриваются основные функциональные элементы компьютера и рассказывается, как они объединяются в единую систему, а также освещается роль системного программного обеспечения, перечисляются базовые аспекты оценки его эффективности и производительности. Кроме того, глава содержит краткий экскурс в историю разработки и создания компьютеров.

Глава 2 посвящена таким вопросам, как машинные команды, технологии адресации и порядок выполнения команд. Вводится понятие арифметики дополнений до двух, без понимания которого невозможно обсуждение процесса формирования исполнительных адресов. При обсуждении циклов, подпрограмм, принципов элементарного программирования ввода-вывода, сортировки и операций со связными списками используются примеры программ на уровне машинных команд, написанные на универсальном языке ассемблера.

Глава 3 иллюстрирует рассмотренные в главе 2 концепции конкретными примерами их реализации на основе трех коммерческих процессоров: ARM, Motorola 68000 и Pentium. В основу процессора ARM положена архитектура RISC, процессора Motorola 68000 — архитектура CISC, а в процессоре Pentium реализована наиболее успешная в коммерческом отношении архитектура, которая объединяет элементы двух предыдущих архитектур, являющихся базовыми. Материал главы разделен на три независимые части. В каждой из них содержатся все примеры из главы 2, реализованные в контексте конкретного процессора. Поэтому достаточно прочесть только одну из трех частей. Если курс сопровождается лабораторными

экспериментами с использованием одного из трех процессоров, параллельно изучению главы 2 можно изучать соответствующую часть главы 3.

Организации ввода-вывода посвящена глава 4. В ней рассматриваются способы синхронизации передаваемых данных и ряд сложных структур ввода-вывода, подробно освещаются механизмы обработки прерываний, в том числе роль программных прерываний в работе операционной системы, и технология прямого доступа к памяти, описываются протоколы и стандарты шин (на примерах шин PCI, SCSI и USB)

Микросхемы памяти, и в частности SDRAM, Rambus, флэш-память, обсуждаются в главе 5. Рассматриваются такие способы расширения полосы пропускания основной памяти, как использование кэш-памяти и систем памяти, состоящих более чем из одного модуля. Особенно подробно обсуждается технология кэширования, включая возможности повышения ее эффективности. Кроме того, описываются системы виртуальной памяти, механизмы управления памятью и методы быстрого преобразования адресов. Как отдельные компоненты системы памяти рассматриваются магнитные и оптические диски.

В главе 6 речь пойдет об арифметическом устройстве компьютера. Здесь анализируется логическая архитектура аппаратного обеспечения, выполняющего сложение, вычитание, умножение и деление чисел с фиксированной запятой, описываются операции дополнения до двух. Мы также поговорим о принципах функционирования сумматоров с параллельным переносом и высокоскоростных умножителей, в том числе рассмотрим алгоритм Бута с перекодировкой множителя и алгоритм сложения с сохранением переноса. Представление чисел с плавающей запятой производится в контексте стандарта IEEE, даны примеры выполнения операций с такими числами.

Глава 7 начинается с рассмотрения процесса выборки команд и их выполнения процессором на уровне пересылки данных между регистрами. Далее рассказывается о реализации процессора на уровне аппаратного обеспечения и микропрограммирования.

В главе 8 подробно рассказывается об использовании технологий конвейерной обработки команд и о применении некоторых исполнительных блоков в архитектуре высокопроизводительных процессоров. Освещается роль компиляторов и связь между конвейерным выполнением и структурой набора команд. Обсуждаются суперскалярные процессоры, принципы построения которых объясняются на примере процессора UltraSPARC II от Sun Microsystems.

В настоящее время существует гораздо больше процессоров для встраиваемых систем, чем процессоров для компьютеров общего назначения. В них на одной микросхеме интегрированы функции процессора, подсистемы ввода-вывода и таймера, необходимые широчайшему диапазону недорогих устройств. Такие процессоры описаны в главе 9. Кроме того, здесь обсуждаются вопросы системной интеграции, межкомпонентных соединений и программного обеспечения, функционирующего в режиме реального времени.

В главе 10 рассматриваются периферийные устройства и способы их подключения к компьютерам. В ней также описываются типичные устройства ввода-вывода и аппаратное обеспечение, необходимое для поддержки компьютерной графики, наиболее распространенные коммуникационные соединения.

Основная тема главы 11 — эволюция процессорных семейств ARM, Motorola и Intel. Здесь освещаются произошедшие в структуре этих процессоров изменения, направленные на повышение их быстродействия, рассматриваются процессоры PowerPC, SPARC, Alpha и Intel IA-64.

В главе 12 от знакомства с устройством отдельных компьютеров мы перейдем к изучению архитектуры больших систем, где задействуется множество одновременно работающих процессоров. В этой связи будут, в частности, рассмотрены сети, используемые для соединения элементов мультипроцессорных устройств, и технологии синхронизации кешей. Описываются архитектуры систем с общей памятью и с передачей сообщений.

Изменения в пятом издании

При подготовке пятого издания этой книги в ее содержание и структуру был внесен ряд изменений. Перечислим наиболее существенные из них.

- Глава 2 четвертого издания теперь разделена на две части: главы 2 и 3. В главе 2 описываются универсальные команды с примерами типичных задач — как связанных с различного рода вычислениями, так и не требующих каких-либо расчетов. В главе 3 представлены наборы команд процессоров ARM, Motorola 68000 и Pentium, демонстрирующие реализацию базовых концепций двух архитектур: RISC и CISC.
- Более подробно рассматривается роль технологий конвейерной обработки команд и возможность использования нескольких исполнительных блоков в архитектуре процессоров. В качестве примера, демонстрирующего применение этих технологий с целью повышения производительности, представлена архитектура процессора UltraSPARC.
- Добавлена новая глава о встраиваемых процессорных системах. В качестве основы для примеров используется обобщенная структура типичной системы.

Кроме того, в книгу внесено множество других, менее значительных изменений и дополнений, связанных с самыми последними технологическими и архитектурными нововведениями.

Что можно изучить за один семестр

Как уже было отмечено, эта книга рекомендуется в качестве учебного пособия для студентов компьютерных специальностей по курсу организации вычислительных систем. Здесь содержится более чем достаточно материала, предназначенного для изучения в течение одного семестра. Основная информация изложена в главах 1–8. Для тех, кто еще не знаком с методами синтеза логических схем, в приложении А приведен базовый материал по этой теме. Студентам необходимо с ним ознакомиться в начале курса или хотя бы до того, как они приступят к изучению материала главы 4.

Главы 9–12 также содержат много полезного материала, из которого преподаватель может выбрать дополнительные темы, если позволит время. Особенно желательно найти время для анализа встраиваемых систем, о которых рассказывается в главе 9, а также для обсуждения рассмотренного в главе 10 аппаратного обеспечения, имеющегося в большинстве современных персональных компьютеров.

Благодарности

Мы выражаем искреннюю признательность всем, кто принял участие в подготовке пятого издания настоящей книги. Гейл Бургес и Келли Чейн помогли нам в работе над рукописью. Алекс Гербик, Франк Шу и Роберт Лу оказали неоценимую помощь в создании большей части примеров. Наши коллеги Тарек Абделрахман, Стивен Браун, Пол Чау, Глен Гулак и Джонатан Роуз высказали ряд конструктивных замечаний. Мы особо благодарны Стивену и Тареку, которые помогли нам уточнить ряд важных деталей. Наши рецензенты — Гойко Бабик из Университета штата Огайо, Натаниел Девис, преподаватель Политехнического института и Университета штата Вирджиния, Хосе Фортес из Университете имени Пердью штата Индиана, Санг Ху из Университета города Сан-Франциско, Али Херсон из Университета штата Пенсильвания, Лизи Кериан Джон из Университета города Остин в штате Техас, Стефан Лью из Университета имени Альберта Людвига во Фрайбурге, Фабризио Ломбардии из Северо-Западного университета, Вейн Лаукс из Университета города Ватерлоо, Прасант Мохапатра из Университета штата Айова, Дениел Табак из Университета имени Джоржа Мэйсона, Джон Вейлуис из Политехнического института имени Ренеселаера города Троя в штате Нью-Йорк — внесли множество ценных предложений и высказали ряд критических замечаний. Мы благодарим Эли Вранешича за разрешение поместить на обложку книги копию его картины «Осень в Гайд-парке» (Fall in High Park). Он написал ее с помощью компьютера. Мы высоко ценим поддержку и помощь нашего редактора Кэтрин Филдс Шалтс, а также ее коллег из издательства McGraw-Hill, в частности Келли Ватчер, Мишель Фломенхофт, Кала Грехем, Бетси Джонс, Рик Ноел, Хезер Сабо и Кристин Уолкер.

Карл Хамахер
Звонко Вранешич
Сафват Заки

Глава 1

Базовая структура компьютеров

- Архитектура компьютера
- Машинные команды и их выполнение
- Программное обеспечение компьютеров
- Производительность компьютерных систем
- История развития вычислительной техники

Основная тема настоящего издания — устройство компьютера. Здесь описываются конструкции и функции различных элементов компьютеров, предназначенных для хранения и обработки информации, рассматриваются компоненты компьютера, которые получают информацию от внешних источников и отсылают результаты вычислений внешним приемникам данных. Большая часть материала книги посвящена *аппаратному обеспечению компьютеров* и их *архитектуре*. Аппаратное обеспечение компьютера состоит из электронных схем, дисплеев, магнитных и оптических устройств для хранения информации, электромеханического оборудования и средств коммуникации. Архитектура компьютера включает спецификацию набора команд и аппаратные компоненты, реализующие эти команды.

В книге обсуждается множество аспектов аппаратных и программных компонентов компьютерных систем. Для наиболее полного освоения и правильного понимания компьютерных систем необходимо учитывать и аппаратные, и программные аспекты каждого их компонента.

Эта глава может служить введением в целый ряд аппаратных и программных концепций и распространенных технологий. В ней приводится общий обзор фундаментальных аспектов изучаемого предмета. Все эти аспекты подробно рассматриваются в следующих главах книги.

1.1. Типы компьютеров

Для начала давайте определим, что такое *цифровой компьютер* или просто *компьютер*. Согласно простейшей интерпретации этого термина, современный компьютер представляет собой электронное вычислительное устройство, которое принимает

дискретную входную информацию, обрабатывает ее в соответствии со списком хранящихся внутри нее команд и генерирует результирующую выходную информацию. Упомянутый список команд называется *компьютерной программой*, а место, где он хранится, — *памятью* компьютера.

Существующие в настоящее время типы компьютеров очень многочисленны и разнообразны; они различаются размерами, стоимостью, вычислительной мощью и назначением. Наиболее распространенным типом компьютеров являются *персональные компьютеры*, широко используемые как дома, так и в учебных заведениях, офисах всевозможных компаний. *Настольные компьютеры* — наиболее популярная форма персональных компьютеров. У настольного компьютера имеются устройства для обработки и хранения данных, дисплей и звуковые выходные устройства, а также клавиатура, располагающаяся на рабочем столе. Устройствами для хранения данных являются жесткие диски, CD-ROM и дискеты. *Портативным компьютером* (*ноутбуком*) называется компактная версия персонального компьютера, в которой все компоненты размещаются в одном блоке, имеющем размер небольшого тонкого портфеля. *Рабочие станции* с графическими входными и выходными устройствами, характеризующиеся высокой разрешающей способностью и имеющие размер настольных компьютеров, обладают значительно большей вычислительной мощью, нежели персональные компьютеры. Они часто используются при выполнении инженерных расчетов, в первую очередь для решения задач автоматизированного проектирования.

Наряду с рабочими станциями существует еще целый спектр больших и очень мощных компьютерных систем — от *корпоративных серверов* и *серверов*, находящихся в нижней части этого спектра, до *суперкомпьютеров*, относящихся к его вершине. Корпоративные серверы и *мэйнфреймы* используются для обработки деловых данных в средних и крупных корпорациях, которым необходимы значительно большая вычислительная мощь и емкость запоминающих устройств, чем могут обеспечить рабочие станции. Серверы содержат устройства для хранения баз данных и могут обрабатывать большое количество запросов. Они широко используются в сфере образования, в бизнесе и различных некоммерческих организациях. Запросы к серверам и их ответы часто транспортируются с помощью коммуникационных средств Интернета. В настоящее время всемирная сеть и связанные с ней серверы являются основным источником всех типов информации мирового уровня. Коммуникационные средства Интернета представляют собой сложный комплекс высокоскоростных оптоволоконных магистральных линий, к которым с помощью телевизионных кабелей и телефонных линий подсоединяются учебные заведения, всевозможные компании и организации, а также дома и квартиры индивидуальных пользователей.

Суперкомпьютеры ипредназначены для проведения крупномасштабных числовых вычислений, необходимых таким приложениям, как, скажем, метеорологические системы или системы для конструирования самолетов и имитационного моделирования. Функциональные блоки (в том числе процессорные комплексы) корпоративных систем, серверов и суперкомпьютеров могут состоять из множества отдельных и часто очень больших устройств.

1.2. Функциональная структура компьютера

Как следует из рис. 1.1, компьютер состоит из пяти главных, функционально независимых частей: устройство ввода, устройство памяти, арифметико-логическое устройство, устройство вывода и устройство управления. Устройство ввода принимает через цифровые линии связи закодированную информацию от операторов, электромеханических устройств типа клавиатуры или от других компьютеров сети. Полученная информация либо сохраняется в памяти компьютера для последующего применения, либо немедленно используется арифметическими и логическими схемами для выполнения необходимых операций. Последовательность шагов обработки определяется хранящейся в памяти программой. Полученные результаты отправляются обратно, во внешний мир, посредством устройства вывода. Все эти действия координируются блоком управления. На рис. 1.1 намеренно не показаны связи между функциональными устройствами. Объясняется это тем, что такие связи могут быть по-разному реализованы. Как именно, вы поймете несколько позже. Арифметические и логические схемы в комплексе с главными управляющими схемами называют *процессором*, а все вместе взятое оборудование для ввода и вывода часто называют *устройством ввода-вывода* (input-output unit).

Теперь обратимся к обрабатываемой компьютером информации. Ее удобно разделять на две основные категории: команды и данные. *Команды*, или *машинные команды*, — это явно заданные инструкции, которые:

- управляют пересылкой информации внутри компьютера, а также между компьютером и его устройствами ввода-вывода;
- определяют подлежащие выполнению арифметические и логические операции.

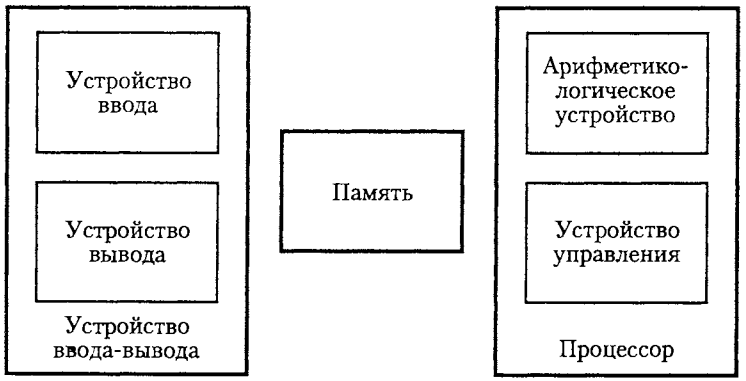


Рис. 1.1. Базовые функциональные устройства компьютера

Список команд, выполняющих некоторую задачу, называется *программой*. Обычно программы хранятся в памяти. Процессор по очереди извлекает команды программы из памяти и реализует определяемые ими операции. Компьютер полностью управляется *хранимой программой*, если не считать возможность внешнего вмешательства оператора и подсоединенных к машине устройств ввода-вывода.

Данные — это числа и закодированные символы, используемые в качестве операндов команд. Однако термин «данные» часто используется для обозначения любой цифровой информации. Согласно этому определению, сама программа (то есть список команд) также может считаться данными, если она обрабатывается другой программой. Примером обработки одной программой другой является *компиляция исходной программы*, написанной на языке высокого уровня, в список машинных команд, составляющих программу на машинном языке, которая называется *объектной программой*. Исходная программа поступает на вход компилятора, который транслирует ее в программу на машинном языке.

Информация, предназначенная для обработки компьютером, должна быть закодирована, чтобы иметь подходящий для компьютера формат. Современное аппаратное обеспечение в большинстве своем основано на цифровых схемах, у которых имеется только два устойчивых состояния, ON и OFF (см. приложение А). В результате кодирования любое число, символ или команда преобразуется в строку двоичных цифр, называемых *битами*, каждый из которых имеет одно из двух возможных значений: 0 или 1. Для представления чисел (как станет ясно из глав 2 и 6) обычно используется позиционная двоичная нотация. Иногда применяется *двоично-десятичный* формат (Binary-Coded Decimal, BCD), в соответствии с которым каждая десятичная цифра кодируется отдельно, с помощью четырех бит.

Буквы и цифры также представляются посредством двоичных кодов. Для них разработано несколько разных схем кодирования. Наиболее распространенными считаются схемы ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией), где каждый символ представлен 7-битовым кодом, и EBCDIC (Extended Binary Coded Decimal Interchange Code — расширенный двоично-десятичный код для обмена информацией), в котором для кодирования символа используется 8 бит. Более подробное описание двоичной нотации и схем кодирования приведено в приложении Д.

1.2.1. Устройство ввода

Компьютер принимает кодированную информацию через устройство ввода, задачей которого является чтение данных. Наиболее распространенным устройством ввода является клавиатура. Когда пользователь нажимает клавишу, соответствующая буква или цифра автоматически преобразуется в определенный двоичный код и по кабелю пересылается либо в память, либо процессору.

Существует и ряд других устройств ввода, среди которых джойстики, трекболы и мыши. Они используются совместно с дисплеем в качестве графических входных устройств. Для ввода звука могут использоваться микрофоны. Воспринимаемые ими звуковые колебания измеряются и конвертируются в цифровые коды для хранения и обработки. Подробное описание звуковых устройств и принципов их функционирования вы найдете в главе 10.

1.2.2. Блок памяти

Задачей блока памяти является хранение программ и данных. Существует два класса запоминающих устройств, а именно первичные и вторичные. *Первичное запоминающее устройство* (primary storage) — это память, быстродействие которой

определяется скоростью работы электронных схем. Пока программа выполняется, она должна храниться в первичной памяти. Эта память состоит из большого количества полупроводниковых ячеек, каждая из которых может хранить один бит информации. Ячейки редко считываются по отдельности — обычно они обрабатываются группами фиксированного размера, называемыми *словами*. Память организована так, что содержимое одного слова, содержащего n бит, может записываться или считываться за одну базовую операцию.

Для облегчения доступа к словам в памяти с каждым словом связывается отдельный *адрес*. Адреса — это числа, идентифицирующие конкретные местоположения слов в памяти. Для того чтобы прочитать слово из памяти или записать его в таковую, необходимо указать его адрес и задать управляющую команду, которая начнет соответствующую операцию.

Количество битов в каждом слове часто называют *длиной машинного слова*. Обычно слово имеет длину от 16 до 64 бит. Одним из факторов, характеризующих класс компьютера, является емкость его памяти. Малые машины обычно могут хранить лишь несколько десятков миллионов слов, тогда как средние и большие машины обычно способны хранить сотни миллионов слов. Типичными единицами измерения количества обрабатываемых машиной данных являются слово, несколько слов или часть слова. Как правило, за время одного обращения к памяти считывается или записывается только одно слово.

Во время выполнения программа должна находиться в памяти. Команды и данные должны записываться в память и считываться из памяти под управлением процессора. Исключительно важна возможность предельно быстрого доступа к любому слову памяти. Память, к любой точке которой можно получить доступ за короткое и фиксированное время, называется *памятью с произвольным доступом* (Random-Access Memory, RAM). Время, необходимое для доступа к одному слову, называется *временем доступа к памяти*. Это время всегда одинаково, независимо от того, где располагается нужное слово. Время доступа к памяти в современных устройствах RAM составляет от нескольких наносекунд до 100. Память компьютера обычно представляет собой *иерархическую структуру*, состоящую из трех или четырех уровней полупроводниковых RAM-элементов с различной скоростью и разным размером. Наиболее быстродействующим типом RAM-памяти является *кэш-память* (или просто *кэш*). Она напрямую связана с процессором и часто находится на одном с ним интегрированном чипе, благодаря чему работа процессора значительно ускоряется. Память большей емкости, но менее быстрая, называется *основной памятью* (main memory). Далее в этой главе процесс доступа к информации в памяти описывается подробнее, а в главе 5 детально рассматриваются принципы ее функционирования и вопросы, связанные с производительностью.

Первичные запоминающие устройства являются исключительно важными коипонентами для компьютера, но они довольно дороги. Поэтому компьютеры оборудуются дополнительными, более дешевыми *вторичными запоминающими устройствами*, используемыми для хранения больших объемов данных и большого количества программ. В настоящее время таких устройств имеется достаточно много. Но наиболее широкое распространение получили *магнитные диски*, *магнитные ленты* и *оптические диски* (CD-ROM). Эти устройства также описываются в главе 5.

1.2.3. Арифметико-логическое устройство

Большинство компьютерных операций выполняется в *арифметико-логическом устройстве* (*АЛУ*) процессора. Рассмотрим типичный пример. Предположим, нам нужно сложить два находящихся в памяти числа. Эти числа пересылаются в процессор, где АЛУ выполняет их сложение. Полученная сумма может быть записана в память или оставлена в процессоре для немедленного использования.

Любые другие арифметические или логические операции, в том числе умножение, деление и сравнение чисел, начинаются с пересылки этих чисел в процессор, где АЛУ должно выполнить соответствующую операцию. Когда операнды переносятся в процессор, они сохраняются в высокоскоростных элементах памяти, называемых *регистрами*. Каждый регистр может хранить одно слово данных. Время доступа к регистрам процессора даже меньше времени доступа к самой быстрой кэш-памяти.

Управляющее и арифметико-логическое устройства работают во много раз быстрее, чем все остальные устройства, подключенные к компьютерной системе. Это позволяет одному процессору контролировать множество внешних устройств, таких как клавиатуры, дисплеи, магнитные и оптические диски, сенсоры и механические управляющие устройства.

1.2.4. Блок вывода

Функция блока вывода противоположна функции блока ввода: он направляет результаты обработки в так называемый внешний мир. Типичным примером устройства вывода является *принтер*. Для печати в принтерах используются ударные механизмы, головки, выпрыскивающие струи чернил, или технологии фотокопирования, как в лазерных принтерах. Существуют принтеры, способные печатать до 10 000 строк в минуту. Для механического устройства это огромная скорость, но по сравнению с быстродействием процессора она ничтожно мала.

Некоторые устройства, и в частности графические дисплеи, выполняют одновременно и функцию вывода, и функцию ввода. Поэтому они называются устройствами ввода-вывода.

1.2.5. Блок управления

Устройства памяти, арифметики и логики, ввода и вывода хранят и обрабатывают информацию, а также выполняют операции ввода и вывода. Работу таких устройств нужно как-то координировать. Именно этим и занимается блок управления. Это, если можно так выразиться, нервный центр компьютера, передающий управляющие сигналы другим устройствам и отслеживающий их состояние.

Управление операциями ввода-вывода осуществляется командами программ, в которых идентифицируются соответствующие устройства ввода-вывода и пересылаемые данные. Однако реальные *синхронизирующие сигналы* (timing signals), управляющие пересылкой, генерируются управляющими схемами. Синхронизирующие сигналы — это сигналы, определяющие, когда должно быть выполнено данное действие. Кроме того, посредством синхронизирующих сигналов, генерируемых блоком управления, осуществляется передача данных между процессором

и памятью. Блок управления можно представить себе как отдельное устройство, взаимодействующее с другими частями машины. Но на практике так бывает редко. Большая часть управляющих схем физически распределена по разным местам компьютера. Сигналы, используемые для синхронизации событий и действий всех устройств, передаются по множеству управляющих линий (проводов).

В целом, функционирование компьютера можно описать следующим образом.

- Компьютер с помошью блока ввода принимает информацию в виде программ и данных и записывает ее в память.
- Хранящаяся в памяти информация под управлением программы пересылается в арифметико-логическое устройство для дальнейшей обработки.
- Данные, полученные в результате обработки информации, направляются на устройства вывода.
- За все действия, производимые внутри машины, отвечает блок управления.

1.3. Основные концепции функционирования

Как было сказано в разделе 1.2, действиями компьютера управляют инструкции. Для выполнения конкретной задачи в память записывается соответствующая программа, состоящая из множества команд. Команды по очереди пересылаются из памяти в процессор, который их выполняет. Данные, используемые в качестве операндов команд, также хранятся в памяти. Вот пример типичной команды:

```
Add  LOCA,R0
```

Эта команда складывает операнд, хранящийся в памяти по адресу LOCA, с операндом, хранящимся в регистре R0 процессора, и помещает результат в этот же регистр. Исходное содержимое памяти по адресу LOCA не меняется, а содержимое регистра R0 перезаписывается. Данная команда выполняется в несколько этапов. Сначала она пересылается из памяти в процессор. Затем операнд команды считывается из памяти по адресу LOCA и складывается с содержимым регистра R0, после чего результирующая сумма записывается в регистр R0.

В описанной команде Add объединяются две операции: доступ к памяти и операция АЛУ. Во многих современных компьютерах эти два типа операций выполняются с помощью отдельных команд. Такое разделение основывается на соображениях производительности, о которых мы поговорим в главе 8. Приведенная выше команда может быть реализована и в виде двух команд:

```
Load  LOCA,R1
Add   R1,R0
```

Первая из этих команд копирует содержимое памяти по адресу LOCA в регистр процессора R1, а вторая команда складывает содержимое регистров R1 и R0 и помещает сумму в регистр R0. Обратите внимание, что в результате выполнения двух команд исходное содержимое обоих регистров уничтожается, а содержимое памяти по адресу LOCA сохраняется.

Пересылка данных между памятью и процессором начинается с отправки в устройство памяти адреса слова, к которому требуется получить доступ, и выдачи

соответствующих управляющих сигналов. Затем данные пересылаются в память или из памяти.

На рис. 1.2 показано, как соединяются между собой память и процессор. Кроме того, рисунок иллюстрирует несколько важных особенностей функционирования процессора, о которых мы с вами еще не говорили. На нем не показана реальная схема соединений этих компонентов, поскольку пока мы обсуждаем только их функциональные характеристики. Более детально соединение компонентов описывается в главе 7 при рассмотрении конструкции процессора.

Кроме АЛУ и управляющих схем процессор содержит множество регистров, предназначенных для разных целей. В *регистре команды* (Instruction Register, IR) содержится код выполняемой в данный момент команды. Ее результат доступен управляющим схемам, которые генерируют сигналы для управления различными элементами, участвующими в выполнении команды. Еще один специализированный регистр, называемый *счетчиком команд* (Program Counter, PC), служит для контроля за ходом выполнения программы. В нем содержится адрес следующей команды, подлежащей выборке и выполнению. Пока выполняется очередная команда, содержимое регистра PC обновляется — в него записывается адрес следующей команды. Говорят, что регистр PC *указывает* на команду, которая должна быть выбрана из памяти. Кроме регистров IR и PC на рис. 1.2 показано *n регистров общего назначения*, от R_0 до R_{n-1}. Для чего они нужны, объясняется в главе 2.

Наконец, еще два регистра обеспечивают взаимодействие с памятью. Это *регистр адреса* (Memory Address Register, MAR) и *регистр данных* (Memory Data Register, MDR). В регистре MAR содержится адрес, по которому производится обращение к памяти, а в регистре MDR — данные, которые должны быть записаны в память или прочитаны из таковой по этому адресу.

Рассмотрим типичный процесс выполнения программы компьютером. Программа располагается в памяти, куда обычно попадает через входное устройство. Ее выполнение начинается с записи в регистр PC адреса первой команды. Содержимое этого регистра пересылается в регистр MAR, а в память направляется управляющий сигнал Read. Когда истекает время, необходимое для доступа к памяти, адресуемое слово (в данном случае — первая команда программы) считывается из памяти и загружается в регистр MDR. Затем содержимое регистра MDR пересылается в регистр IR. Команда готова к декодированию и выполнению.

Если команда требует, чтобы АЛУ выполнило определенную операцию, для нее необходимо получить операнды. Операнд, располагающийся в памяти (он может находиться и в регистре общего назначения), нужно сначала из таковой извлечь, переслав его адрес в регистр MAR и инициализировав цикл Read. После пересылки из памяти в регистр MDR операнд будет направлен в АЛУ. Аналогичным образом туда же будут переданы и остальные необходимые команде операнды, после чего АЛУ сможет выполнить требуемую операцию. Если результат должен быть сохранен в памяти, он будет записан в регистр MDR. Затем адрес, по которому его нужно записать в память, будет помещен в регистр MAR, после чего будет инициирован цикл Write. В какой-то момент в ходе выполнения текущей инструкции содержимое регистра PC увеличивается, и он начинает указывать на следующую подлежащую выполнению инструкцию. Другими словами, как только завершится выполнение текущей инструкции, можно будет приступать к выборке следующей.

Компьютер не только пересылает данные между памятью и процессором, но и принимает их от входных устройств, а также отсылает выходным устройствам. Поэтому среди машинных команд имеются и команды для выполнения операций ввода-вывода.

Если возникает необходимость срочно обслужить некоторое устройство (например, когда устройство мониторинга в автоматизированном промышленном процессе обнаружит опасную ситуацию), нормальное выполнение программы может быть прервано. Для того чтобы немедленно отреагировать на эту ситуацию, компьютер должен прервать выполнение текущей программы. С этой целью устройство генерирует сигнал прерывания. *Прерывание* (interrupt) — это запрос, поступающий от устройства ввода-вывода, с требованием предоставить ему процессорное время. Для обслуживания этого устройства процессор выполняет соответствующую *программу обработки прерывания*. А поскольку ее выполнение может изменить внутреннее состояние процессора, перед обслуживанием прерывания нужно сохранить его состояние в памяти. Обычно в ходе этой операции сохраняется содержимое регистра PC, регистров общего назначения и некоторая управляющая информация. По завершении работы программы обработки прерывания состояние процессора восстанавливается и прерванная программа продолжает свою работу. Процессор со всеми его элементами (рис. 1.2) обычно реализуется в виде одной микросхемы, на которой располагается как минимум одно устройство кэш-памяти. Такие чипы называются VLSI (VLSI — аббревиатура от Very Large Scale Integration, что переводится как очень крупномасштабная интеграция).

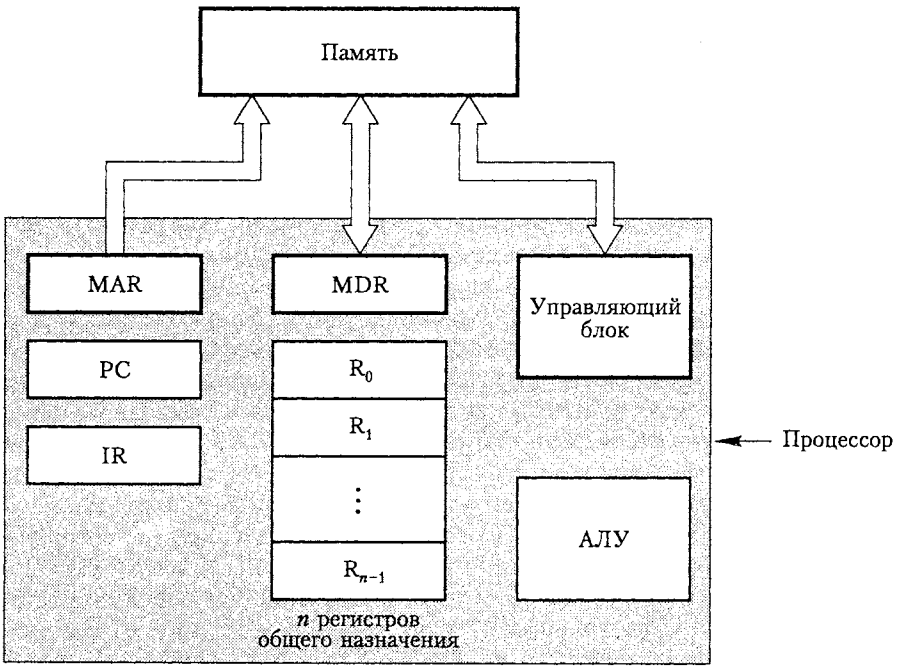


Рис. 1.2. Соединения между процессором и памятью

1.4. Структура шины

До сих пор речь шла о функциях отдельных частей компьютера. Однако для того чтобы составить действующую систему, эти части должны быть соединены между собой определенным образом. Способов их соединения существует очень много, но мы рассмотрим лишь простейшие и самые распространенные из них.

Компьютер сможет работать с достаточной скоростью лишь при условии, что будет организован таким образом, чтобы полное слово данных обрабатывалось им за указанное время. Когда слово данных пересылается между устройствами, параллельно перемещаются и все его биты. Каждый бит пересылается по своему проводу (линии), так что для пересылки слова требуется несколько параллельных линий. Группа линий, образующая соединение между несколькими устройствами, называется *шиной* (bus). Наряду с линиями, по которым пересылаются данные, шина содержит линии для передачи адреса и управляющих сигналов.

Для соединения нескольких функциональных устройств компьютера проще всего использовать *общую шину* (single bus), как показано на рис. 1.3. К этой шине подсоединяются все устройства компьютера. Поскольку за один раз по шине может пересылаться только одно слово данных, в каждый конкретный момент шину могут использовать только два устройства. Для организации процесса параллельной обработки нескольких запросов используются линии управления шиной. Главным достоинством архитектуры с общей шиной является ее низкая стоимость и гибкость в отношении подключения периферийных устройств. При наличии в системе нескольких шин возможно одновременное выполнение нескольких операций пересылки данных, благодаря чему такая система работает быстрее, но и стоимость ее выше.

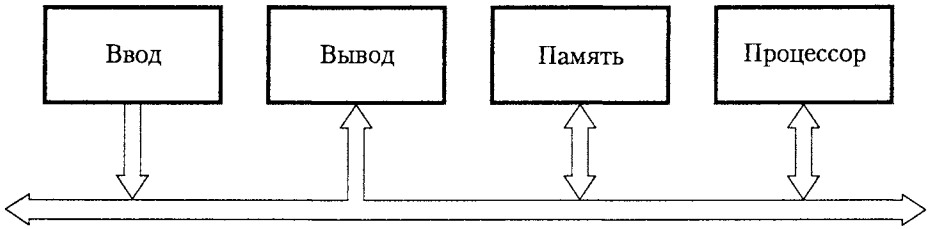


Рис. 1.3. Архитектура системы с общей шиной

Подсоединенные к шине устройства могут заметно отличаться друг от друга по скорости функционирования. Некоторые электромеханические устройства, в том числе клавиатуры и принтеры, работают относительно медленно. Значительно выше скорость работы, скажем, магнитных и оптических дисков. А память и процессор функционируют со скоростью электронных схем, благодаря чему являются самыми быстрыми частями компьютера. Поскольку все эти три типа устройств могут взаимодействовать между собой через шину, необходим такой механизм пересылки данных, который не ограничивал бы скорость обмена информацией между любыми двумя устройствами скоростью более медленного из них и сглаживал бы разницу в скорости работы процессора, памяти и внешних устройств.

Самый распространенный подход к решению этой задачи основан на использовании *буферных регистров*, которые встраиваются во внешние устройства для хранения получаемой ими информации. Для примера давайте рассмотрим процесс передачи кода символа от процессора принтеру. Процессор пересылает данные по шине в буфер принтера. Поскольку буфер представляет собой электронный регистр, пересылка выполняется очень быстро. Когда буфер будет заполнен, принтер начнет печатать, и вмешательство процессора больше не потребуется. Шина и процессор освобождаются для другой работы, которая может выполняться одновременно с печатью символа, хранящегося в буфере принтера. Таким образом, использование буферных регистров сглаживает различия в скорости функционирования процессора, памяти и устройств ввода-вывода и предотвращает блокирование высокоскоростного процессора медленными устройствами на все время выполнения операций ввода-вывода. Процессор может быстро переключаться от одного устройства к другому, обслуживая их параллельно.

1.5. Программное обеспечение

Для того чтобы пользователь мог запустить прикладную программу, в памяти компьютера должно уже содержаться некоторое системное программное обеспечение. *Системное программное обеспечение* — это набор программ, предназначенных для выполнения следующих функций:

- получение и интерпретация команд пользователя;
- ввод и редактирование прикладных программ, их сохранение в файлах на вторичных запоминающих устройствах;
- управление процессом сохранения файлов на вторичных запоминающих устройствах и извлечение их с указанных устройств;
- запуск стандартных прикладных программ, таких как текстовые процессоры, электронные таблицы или игры, с данными, которые предоставляются пользователем;
- управление устройствами ввода-вывода для получения входной информации и вывода выходных данных;
- трансляция исходного кода программ, подготовленных ранее пользователем, в объектные модули, состоящие из машинных команд;
- компоновка пользовательских прикладных программ со стандартными библиотечными подпрограммами (например, выполняющими числовые вычисления) и запуск результирующих программ.

Таким образом, системное программное обеспечение отвечает за координирование всех операций, выполняемых в компьютерной системе. В этом разделе мы рассмотрим его важнейшие аспекты.

Прикладные программы обычно пишутся на языках программирования высокого уровня, в том числе на C, C++, Java и FORTRAN, позволяющих программисту задать действия, которые должна выполнить программа (скажем, математические вычисления или обработку строк текста). Такие операции описываются в формате, не зависящем от типа компьютера, который будет выполнять программу.

Программисту, использующему язык высокого уровня, не нужно знать машинные команды и особенности их использования. Специальная системная программа, называемая *компилятором*, транслирует программу на языке высокого уровня в программу на машинном языке, состоящую из таких команд, как Add и Load, о которых рассказывалось в разделе 1.3.

Еще одна важная системная программа, которой пользуются все программисты, называется *текстовым редактором*. Она предназначена для ввода и редактирования прикладных программ. Пользователь такой программы с помощью клавиатуры вводит и редактирует инструкции исходного текста программы и накапливает их в *файле*. Файл — это просто последовательность буквенно-цифровых символов или двоичных данных, которая сохраняется в памяти или на вторичном запоминающем устройстве. К файлу можно обращаться по заданному пользователем имени.

В этой книге мы не станем подробно рассматривать компиляторы, редакторы и файловые системы, а сосредоточим внимание на ключевом компоненте системного программного обеспечения, называемом *операционной системой* (*ОС*). Это большая программа, а точнее, целый набор программ, используемый для управления взаимодействием различных устройств компьютера при выполнении прикладных программ. Компоненты операционной системы отвечают за предоставление прикладным программам ресурсов компьютера — основной памяти и памяти на магнитных дисках, устройств ввода-вывода и т. д.

Чтобы понять назначение и основные принципы функционирования операционной системы, давайте рассмотрим систему с одним процессором, одним диском и одним принтером. Сначала мы обсудим процесс выполнения прикладной программы. Когда он станет вам понятен, мы сможем поговорить о том, как операционная система организует одновременное выполнение нескольких прикладных программ. Предположим, у нас имеется уже откомпилированная прикладная программа, сохраненная в виде машинных команд на диске. Первым делом файл, в котором хранится эта программа, нужно переслать в память. Как только это будет сделано, начнется выполнение программы. Допустим, что в функции программы входит чтение файла данных с диска в основную память, выполнение определенных вычислений с этими данными и печать результатов. Когда выполняемой программе потребуется файл данных, программа попросит операционную систему переслать ей таковой с диска в память. Операционная система выполнит этот запрос и вернет управление прикладной программе, которая перейдет к вычислениям. Когда вычисления будут завершены и придет время печати результатов, прикладная программа снова направит запрос операционной системе. В ответ будет осуществлена соответствующая программа операционной системы, которая обеспечит отправку нужных данных на принтер.

Итак, вы видите, что при выполнении прикладной программы управление постоянно передается то ей, то программам операционной системы. Процесс поочередного использования процессора можно проиллюстрировать линейной диаграммой, показанной на рис. 1.4. В течение времени от момента t_0 до момента t_1 одна из программ операционной системы инициирует загрузку прикладной программы с диска в память, дожидается завершения процесса загрузки, а затем

передает управление прикладной программе. Аналогичные процессы происходят с момента t_2 до момента t_3 и с момента t_4 до момента t_5, когда операционная система считывает файл данных с диска в основную память и когда она печатает результаты. После момента времени t_5 операционная система может загрузить и выполнить другую прикладную программу.

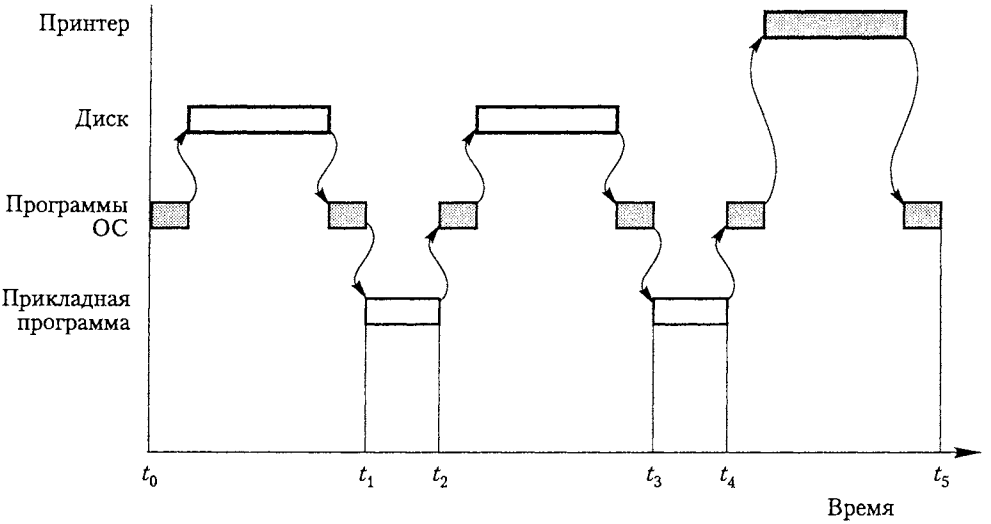


Рис. 1.4. Поочередное использование процессора прикладной программой и программами операционной системы

А теперь давайте рассмотрим способ более эффективного использования ресурсов компьютера, заключающийся в параллельном выполнении нескольких программ. Обратите внимание, что в течение отрезка времени от момента t_4 до момента t_5 ни процессор, ни диск не заняты никакой работой. В это время работает только принтер, и операционная система могла бы загрузить в память следующую программу. Аналогичным образом, в промежуток времени от t_0 до t_1 операционная система могла бы печатать результаты, сгенерированные предыдущей программой (пока текущая программа загружается с диска). Именно так операционная система управляет параллельным выполнением нескольких прикладных программ, обеспечивая поочередное использование ими ресурсов компьютера. Такая схема параллельного выполнения программ называется *многозадачностью*.

1.6. Производительность

Одним из важнейших критериев оценки производительности компьютера является то, насколько быстро он выполняет программы. Скорость выполнения программ компьютером зависит от конструкции его аппаратного обеспечения и от набора команд машинного языка. Поскольку программы обычно пишутся на языке высокого уровня, производительность зависит еще и от того, насколько удачно

компилятор переводит их на машинный язык. Таким образом, для достижения максимальной производительности нужно, чтобы компилятор, набор машинных команд и аппаратное обеспечение компьютера имели оптимальную структуру. В детали построения компиляторов мы в этой книге вдаваться не будем, а вот структуру набора команд и архитектуру аппаратного обеспечения рассмотрим самым подробным образом.

В разделе 1.5 рассказывалось о том, как операционная система координирует различные действия нескольких параллельно функционирующих программ (выполнение программы, операции с данными на дисках, печать) для достижения максимальной эффективности использования ресурсов компьютера. *Общее время выполнения программы* (elapsed time), на рис. 1.4 равное $t_5 - t_0$, является мерой производительности всей компьютерной системы. Оно зависит от быстродействия процессора, диска и принтера. Рассматривая производительность процессора, мы должны учитывать только те периоды, в течение которых он активен. На рис. 1.4 они соответствуют обозначениям «Прикладная программа» и «Программы ОС». Суммарное время выполнения прикладных программ и программ операционной системы называется *процессорным временем*, необходимым для выполнения программы. Далее мы определим несколько ключевых параметров, влияющих на процессорное время, и укажем главы, в которых они обсуждаются. Советуем читателям при изучении материала последующих глав постоянно держать в голове общую картину производительности компьютерной системы.

Равно как общее время выполнения программы зависит от скорости работы всех устройств компьютерной системы, процессорное время зависит от аппаратного обеспечения, участвующего в выполнении конкретных машинных команд. Аппаратное обеспечение включает процессор и память, обычно соединенные шиной (рис. 1.3). Архитектура системы с общей шиной, представленная на рис. 1.5, отличается от приведенной на рис. 1.3 лишь наличием кэш-памяти, включенной в состав процессорного устройства. Давайте рассмотрим поток команд программы и данных между памятью и процессором. По мере выполнения программы ее команды по одной выбираются из памяти и по шине пересылаются в процессор, а их копии помещаются в кэш. Когда для выполнения команды требуются данные, расположенные в основной памяти, они также пересылаются в процессор, а их копии помещаются в кэш. Если позднее та же команда или элемент данных потребуются еще раз, они будут прочитаны не из основной памяти, а из кэша.

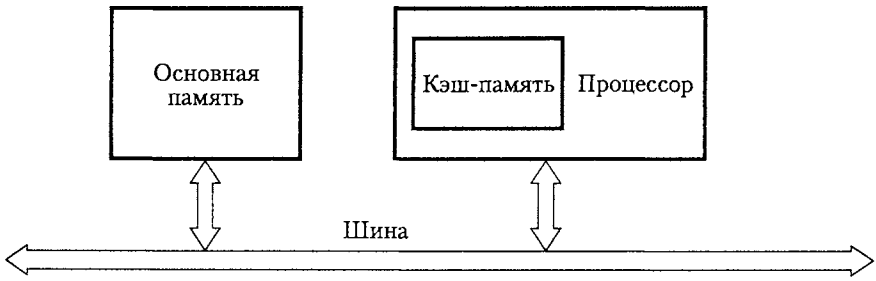


Рис. 1.5. Расположение кэш-памяти на микросхеме процессора

Процессор и относительно небольшая кэш-память могут располагаться на одном интегрированном чипе. Внутренняя скорость выполнения команд таким чипом очень высока — она гораздо выше, чем скорость выборки команд и данных из памяти. Поэтому программа будет выполняться быстрее при условии минимизации количества команд и объема данных, перемещаемых между процессором и основной памятью. Для этого и предназначается кэш процессора. Возьмем, к примеру, многократное выполнение одной и той же группы команд в течение короткого промежутка времени, как это часто бывает в программных циклах. Если эти команды находятся в кэше, их можно быстро извлекать оттуда в течение всего времени их по многу раз повторяющегося выполнения. Сказанное касается и многократно используемых данных. О структуре, функционировании и производительности основной памяти и кэша подробно рассказывается в главе 5.

1.6.1. Частота процессора

Управление процессором осуществляется с помощью сигналов, которые называются *тактовыми импульсами* (clock) и выдаются через фиксированные интервалы времени. Промежуток времени между двумя тактовыми импульсами составляет *тактовый цикл* (clock cycle), или просто *такт*. Для выполнения машинной команды процессор разделяет ее на последовательность базовых шагов, каждый из которых может быть выполнен за один такт. Длительность одного тактового цикла P является важнейшим параметром, определяющим производительность процессора. Обратное ей значение $R = 1/P$, называемое *тактовой частотой* (clock rate) процессора, измеряется количеством тактов в секунду. Процессоры, используемые в современных персональных компьютерах и рабочих станциях, имеют тактовую частоту от нескольких сотен миллионов до миллиарда тактов в секунду. *Герц* (*Гц*) — единица измерения тактовой частоты — равен одному такту в секунду. Слову «миллион», как известно, соответствует префикс *Мега* (*М*), а слову «миллиард» — префикс *Гига* (*Г*). Таким образом, частота, равная 500 миллионам тактов в секунду, обозначается как 500 мегагерц (МГц), а частота, равная 1250 миллионам тактов в секунду, — как 1,25 гигагерц (ГГц). Соответствующие тактовые периоды равны 2 и 0,8 нс.

1.6.2. Основная формула вычисления производительности

Одним из компонентов общего времени выполнения программы является процессорное время. Предположим, что для реализации программы, написанной на одном из языков высокого уровня, требуется T с процессорного времени. Компилятор генерирует соответствующую объектную программу на машинном языке. Допустим, что для полного выполнения такой программы нужно произвести N команд машинного языка. Значение N — это количество машинных команд, которые будут реально выполнены; оно не обязательно равняется количеству команд в объектной программе. Некоторые команды могут выполняться более одного раза, например, в том случае, если они расположены внутри программного цикла. Другие команды могут вообще не выполняться, что зависит от входных данных.

Предположим, что среднее количество базовых шагов, необходимых для выполнения одной машинной команды, равняется S и что каждый базовый шаг производится за один такт процессора. Если тактовая частота равна R тактам в секунду, время выполнения программы составит

$$T = \frac{N \times S}{R} \tag{1.1}$$

Это равенство часто называют *основной формулой вычисления производительности.*

Для пользователя приложения параметр T имеет гораздо большее значение, чем параметры N, S и R. Для обеспечения более высокой производительности конструктор компьютера должен искать пути уменьшения значения этого параметра, для чего необходимо предельно уменьшить значения N и S и увеличить значение R. Значение N уменьшается, когда исходная программа компилируется в объектную программу с меньшим количеством команд. Значение S уменьшается, когда процесс выполнения команды состоит из меньшего количества базовых шагов или же если некоторые шаги команд могут выполняться одновременно. С повышением тактовой частоты повышается значение R и сокращается время выполнения базового шага команды.

Важно подчеркнуть, что параметры N, S и R отнюдь не являются независимыми друг от друга — изменение одного из них может повлиять на величину другого. Поэтому любое новшество в конструкции процессора повысит производительность компьютера только в том случае, если в результате уменьшится значение параметра T. Процессор с частотой 900 МГц не всегда будет работать быстрее, чем процессор с частотой 700 МГц, поскольку у него может быть другое значение параметра S.

1.6.3. Конвейерная и суперскалярная обработка

В предыдущих разделах мы предполагали, что команды выполняются поочередно. Поэтому считали, что значение параметра S равно общему количеству базовых шагов или тактов процессора, необходимых для выполнения одной машинной команды. Но если команды будут выполняться не строго по очереди, а параллельно, производительность процессора значительно повысится. Такая технология называется *конвейерной обработкой* (pipelining). Рассмотрим команду

```
Add  R1,R2,R3
```

которая суммирует содержимое регистров R1 и R2 и помещает результат в регистр R3. Эта команда выполняется в два этапа: сначала АЛУ прибавляет значение R1 к содержимому R2, а затем записывает полученную сумму в регистр R3. Пока АЛУ выполняет сложение, процессор может уже считывать из памяти следующую команду. Затем, если для выполнения этой команды также требуется АЛУ, ее операнды могут пересылаться во входные регистры АЛУ одновременно с пересылкой в регистр R3 результата команды Add. В идеале одновременно должно выполняться как можно больше шагов команд, так чтобы за каждый тактовый

период производился не один шаг, а одна команда. Конечно, далеко не всем инструкциям достаточно будет одного такта. Но, с теоретической точки зрения, для получения наименьшего значения параметра T значение параметра S должно быть равным 1.

Конвейерная обработка подробно обсуждается в главе 8. Там будет показано, что на практике идеальное значение $S = 1$ по многим причинам недостижимо. И тем не менее конвейерная обработка значительно ускоряет выполнение команд и позволяет приблизить реальное значение параметра S к единице.

Еще более высокой степени параллелизма можно достичь путем реализации в процессоре нескольких конвейеров команд. Речь идет об использовании нескольких функциональных блоков, обеспечивающих параллельное выполнение команд. В таком случае на каждый такт может припадать начало сразу нескольких команд. Такой режим функционирования процессора называется *суперскалярным*. Если в ходе реализации программы он может поддерживаться достаточно долгое время, реальное значение параметра S получится даже меньшим единицы. Конечно, при параллельном выполнении команд должна сохраняться логическая правильность программы, то есть ее конечные результаты должны быть точно такими же, как при последовательном выполнении. Суперскалярную архитектуру имеют многие современные высокопроизводительные процессоры.

1.6.4. Тактовая частота

Существует два способа увеличения тактовой частоты процессора, то есть параметра R. Первый из них заключается в усовершенствовании технологии работы *интегральной схемы*. Усовершенствованная логическая схема работает быстрее, и тем самым сокращается время выполнения базового шага команды. Это позволяет сократить тактовый период P и увеличить тактовую частоту R. Второй способ состоит в уменьшении количества операций, выполняемых за один базовый шаг, что также позволяет сократить тактовый период. Однако в этом случае общее количество операций, необходимых для выполнения команды, остается прежним, а значит, просто увеличивается количество базовых шагов операции.

Увеличение значения параметра R, обусловленное лишь усовершенствованием интегральной схемы, влияет на все аспекты функционирования процессора, за исключением времени доступа к основной памяти. Но при наличии кэша процент обращений к основной памяти достаточно мал. Поэтому за счет усовершенствования интегральной схемы можно добиться значительного повышения производительности. Значение параметра T уменьшается во столько же раз, во сколько раз увеличивается значение параметра R, поскольку значения параметров S и N при этом не меняются. Труднее определить влияние на производительность способа разделения команд на базовые шаги. Этот вопрос обсуждается в главе 8.

1.6.5. Система команд процессора

Чем проще команда, тем меньшее количество базовых шагов необходимо для ее выполнения. Сложные команды могут состоять из большого числа шагов. Если

все команды процессора просты, для выполнения конкретной программной задачи требуется большое количество команд. Это означает, что значение параметра N будет большим, а значение параметра S — маленьким. С другой стороны, если отдельные команды выполняют более сложные операции, для реализации той же задачи требуется меньшее количество команд, а значит, значение N уменьшается, а значение S, наоборот, увеличивается. Какой из двух вариантов лучше, сказать трудно.

Основным фактором, учитываемым при сравнении двух вариантов системы команд процессора, является возможность использования конвейерной обработки. Выше мы указывали, что для процессора с конвейерной обработкой команд значение параметра S приближается к единице, даже если количество базовых шагов, составляющих одну команду, достаточно велико. Это означает, что чем сложнее команды процессора, тем быстрее выполняется программа. Однако эффективную конвейерную обработку все же проще реализовать для простых команд. Поэтому систему команд проектируют таким образом, чтобы она оптимально подходила для конвейерного выполнения, и часто именно этот фактор является решающим.

О проектировании системы команд процессора и о возможных вариантах его архитектуры рассказывается в главе 2. Сравнительные характеристики процессоров с простыми и сложными командами изучены достаточно основательно. Архитектура первого типа получила название RISC (Reduced Instruction Set Computer — компьютер с сокращенным набором команд), а архитектура второго типа — CISC (Complex Instruction Set Computer — компьютер с полным набором команд). В главах 3 и 11 приводятся примеры процессоров обоих типов и обсуждаются преимущества каждого из них. Читателям же следует помнить, что хотя термины RISC и CISC считаются общепринятыми, они не являются названиями четко определенных типов процессоров. Архитектура каждого конкретного процессора представляет собой результат множества компромиссов. Термины RISC и CISC в большей мере относятся к принципам и технологиям построения процессоров, и мы к этим вопросам еще не раз будем возвращаться.

1.6.6. Компилятор

Компилятор преобразует программу, написанную на языке высокого уровня, в последовательность машинных команд. Для уменьшения значения параметра N нам нужен подходящий набор машинных команд и компилятор, способный максимально эффективно его задействовать. *Оптимизирующий компилятор* использует особенности процессора, который будет выполнять программу, для того чтобы предельно сократить величину произведения $N \times S$, определяющую общее количество тактов. Из главы 8 станет ясно, что количество тактов зависит не только от выбора команд, но и от порядка их следования в программе. Компилятор может реорганизовать команды в программе для достижения лучшей производительности. Конечно, такие изменения не должны отразиться на результатах вычислений.

Хотя по отношению к процессору компилятор является внешним элементом (и даже может быть создан другим производителем), он должен быть тесно связан с его архитектурой. Но очень часто компилятор и процессор разрабатываются

параллельно, а их создатели для достижения наилучшего результата активно сотрудничают. Их общей целью является сокращение общего количества тактов, необходимых для выполнения программной задачи.

1.6.7. Оценка производительности

Чтобы оценить производительность компьютера, ее необходимо как-то измерить. Руководствуясь показателями производительности, конструкторы компьютеров обычно оценивают эффективность новых элементов и технологий, производители базируют на них маркетинговую политику, а покупатели производят выбор из числа имеющихся в продаже моделей.

В предыдущих разделах предполагалось, что единственным параметром, наиболее точно определяющим производительность компьютера, является время выполнения программы T. Несмотря на концептуальную простоту формулы (1.1), вычислить значение T не так-то просто. Более того, такие параметры, как тактовая частота, а также различные архитектурные параметры компьютера не являются достоверными показателями производительности.

Поэтому производительность компьютеров принято измерять с помощью тестовых программ. Для того чтобы можно было сравнивать производительность разных систем, эти программы должны быть стандартизированы. Показателем производительности является время, в течение которого компьютер выполняет заданный тест. Раньше предпринимались попытки создания искусственных программ, которые могли бы использоваться в качестве стандартных тестов. Однако практика показала, что с их помощью невозможно точно предсказывать скорость выполнения реальных прикладных программ.

В настоящее время общепринятой практикой является использование некоторого набора специально подобранных реальных прикладных программ. Подбором таких приложений занимается некоммерческая организация под названием System Performance Evaluation Corporation (SPEC). Она публикует списки программ для разных прикладных областей и результаты тестирования многих имеющихся на рынке моделей компьютеров. Для компьютеров общего назначения набор тестовых приложений был определен в 1989 году. С тех пор он дважды модифицировался, а его новые версии были опубликованы в 1995 и 2000 годах.

В этот список входят самые разнообразные программы, от игр, компиляторов и приложений баз данных до программ, производящих интенсивные вычисления в области астрофизики и квантовой химии. В каждом случае программа компилируется для тестируемого компьютера и измеряется реальное время ее выполнения на этом компьютере. (Никакая эмуляция не допускается.) Та же самая программа компилируется и выполняется на компьютере, выбранном в качестве эталона. Для теста SPEC95 в качестве такового применяется компьютер SUN SPARCstation 10/40, а для теста SPEC2000 — рабочая станция UltraSPARC10 с процессором UltraSPARC-Iii, тактовая частота которого составляет 300 МГц. Коэффициент производительности SPEC вычисляется по следующей формуле:

$$\text{SPEC-коэффициент} = \frac{\text{Время выполнения на эталонном компьютере}}{\text{Время выполнения на тестируемом компьютере}}$$

Таким образом, SPEC-коэффициент 50 указывает на то, что тестируемый компьютер выполняет данный тест в 50 раз быстрее, чем компьютер UltraSPARC10. Для проведения полного тестирования по очереди компилируются и выполняются все программы из списка SPEC, а затем вычисляется среднее геометрическое полученных результатов. Итоговый SPEC-коэффициент для конкретного компьютера рассчитывается по формуле

$$\text{SPEC-коэффициент} = \left(\prod_{i=1}^{n} \text{SPEC}_i\right)^{\frac{1}{n}}$$

где n — количество программ в тестовом наборе.

Поскольку измеряется реальное время выполнения приложений, SPEC-коэффициент характеризует результат суммарного влияния всех факторов, от которых зависит производительность компьютера, в том числе влияние компилятора, операционной системы, процессора и памяти. Более подробную информацию о тестовых программах SPEC и результатах тестирования вы найдете на web-узле корпорации по адресу www.spec.org.

1.7. Мультипроцессорные и мультикомпьютерные системы

До сих пор мы с вами рассматривали компьютеры с одним процессором. Однако большие компьютерные системы могут содержать достаточно много процессорных устройств. Такие системы называются *мультипроцессорными*. Они могут параллельно выполнять либо несколько различных прикладных задач, либо несколько подзадач одной крупной задачи. Обычно все процессоры имеют доступ ко всей памяти системы, которую в таком случае называют *мультипроцессорной системой с общей памятью*. Высокая производительность таких систем достигается за счет их большой сложности и очень высокой стоимости. Их стоимость определяется не только огромным количеством процессоров и объемом памяти, но и более сложными схемами их внутренних соединений.

Наряду с мультипроцессорными системами используются так называемые *мультикомпьютерные* (*многомашинные*) *системы*, представляющие собой соединенные друг с другом группы компьютеров. Такое соединение также позволяет повысить вычислительную мощь системы. Обычно каждому компьютеру доступна только его собственная память. Для того чтобы при выполнении задачи обменяться данными, компьютеры пересылают друг другу *сообщения* через коммуникационную сеть. Эта особенность отличает их от мультипроцессорных систем с общей памятью, и именно ей они обязаны названием *мультикомпьютерные системы со связью через сообщения*.

1.8. Историческая справка

Появлению 60 лет назад первых компьютеров предшествовала долгая и медленная эволюция механических вычислительных устройств. Об этом вы можете прочесть

во многих книгах, журналах и других источниках. Мы же лишь дадим краткий обзор истории разработки компьютеров.

В течение последних трех столетий, вплоть до середины двадцатого века, для выполнения базовых операций сложения, вычитания, умножения и деления изобретались все более сложные механизмы, состоящие из колесиков, рычагов и блоков. Для автоматического управления последовательностью вычислений сначала использовали перфорированные карты, расположение отверстий в которых определялось механическим путем. Эти карты представляли собой прообраз компьютерных программ. Механические устройства могли вычислять целые математические таблицы логарифмов и тригонометрических функций. Выходные результаты пробивали на картах или печатали на бумаге. Во время второй мировой войны были сконструированы компьютеры, основанные на электромеханических реле, подобных тем, которые использовались в ранних телефонных коммутаторах. Тогда же в Университете штата Пенсильвания был разработан и первый электронный компьютер, основанный на технологии вакуумных ламп, использовавшихся в то время в радиоприемниках и военных радарах. Вакуумные лампы применялись для выполнения логических операций и хранения данных. Эта технология положила начало новой эре электронных цифровых компьютеров.

Все созданные за это время компьютеры в зависимости от технологий, используемых при проектировании и изготовлении процессоров, устройств памяти и систем ввода-вывода, можно отнести к одному из четырех поколений: первое — с 1945 по 1955 год; второе — с 1955 по 1965, третье — с 1965 по 1975, а четвертое с 1975 года по сегодняшний день.

1.8.1. Первое поколение

Столь важная для компьютеров концепция хранимых программ была введена Джоном фон Нейманом. Согласно этой концепции, программы и их данные, как и сейчас, располагались в одной и той же области памяти. Для написания программ использовался язык ассемблера, который затем транслировался в машинный язык для выполнения.

Для реализации логических функций применялась технология вакуумных ламп, обеспечивавшая выполнение базовых арифметических операций за несколько миллисекунд. По сравнению с механическими и электромеханическими машинами на основе реле скорость вычислений увеличилась в в сотни и даже тысячи раз. В компьютерах первого поколения поначалу использовалась память на основе линий задержки, а функции ввода-вывода выполнялись устройствами, похожими на печатные машинки. Затем появилась память на магнитных сердечниках и устройства хранения на магнитных лентах.

1.8.2. Второе поколение

Первые транзисторы были разработаны сотрудниками AT&T Bell Laboratories в начале 1940-х годов. Применение транзисторов, которые очень быстро заменили вакуумные лампы, ознаменовало появление компьютеров второго поколения.

В этих компьютерах уже использовались память на магнитных сердечниках и накопители на магнитных барабанах. Появились языки высокого уровня, и в частности FORTRAN, значительно облегчившие разработку прикладного программного обеспечения. Со временем были изобретены компиляторы для трансляции программ с языков высокого уровня на язык ассемблера, который, в свою очередь, транслировался в машинные коды. В это же время были созданы и процессоры ввода-вывода, функционирующие параллельно с выполнявшим программы центральным процессором, за счет чего увеличивалась общая производительность компьютера. В этот период ведущим производителем компьютерной техники стала компания IBM.

1.8.3. Третье поколение

С появлением технологии объединения множества транзисторов на одном кремниевом чипе, названной технологией интегральных схем, стало возможным создание недорогих, но быстрых процессоров и элементов памяти. Интегральные схемы памяти заменили память на магнитных сердечниках. Начался отсчет эры компьютеров третьего поколения. В этот период было создано множество программных технологий, широко используемых до настоящего времени: микропрограммирование, параллелизм, конвейерная обработка. Программное обеспечение операционных систем позволило совместно использовать ресурсы компьютера несколькими пользовательскими программами. Были разработаны кэш и виртуальная память. Кэш-память представляет основную память для процессора более быстрой, а виртуальная память — намного большей, чем она есть на самом деле. Доминирующими коммерческими продуктами третьего поколения стали мэйнфреймы System 360 от IBM и линия миникомпьютеров PDP от Digital Equipment Corporation.

1.8.4. Четвертое поколение

В начале 1970-х годов развитие технологии производства интегральных схем достигло того этапа, когда стало возможным интегрировать в одном чипе все компоненты процессора и большие фрагменты основной памяти малых компьютеров. Речь идет о технологии производства чипов, содержащих десятки тысяч транзисторов, получила название VLSI (Very Large Scale Integration — очень крупномасштабная интеграция). VLSI позволяет создавать процессоры, состоящие из единственного чипа. Их также называют микропроцессорами. Лидерами этой технологии стали компании Intel, National Semiconductor, Motorola, Texas Instruments и Advanced Micro Devices.

В производстве современных компьютерных систем используются такие архитектурные концепции, как параллельная обработка, конвейерная обработка, кэширование и виртуальная память. Портативные компьютеры, настольные персональные компьютеры и рабочие станции, соединенные локальными и глобальными сетями, а также Интернет стали основными средствами решения различных вычислительных задач. Мэйнфреймы теперь применяются для централизованных вычислений преимущественно в бизнес-приложениях больших компаний.

1.8.5. После четвертого поколения

Иногда самые современные, управляемые с помощью приложений компьютеры называют компьютерами следующего поколения. В последние годы появилась тенденция при именовании каждой новой компьютерной технологии использовать уже не номер поколения, а название, определяющее ее функции. Например: системы с элементами искусственного интеллекта, машины с высокой степенью параллелизма, сильно распределенные системы. Пожалуй, наиболее важной особенностью развития современной компьютерной индустрии является увеличение мощи и доступности настольных компьютеров и широчайшее использование информационных ресурсов Интернета.

1.8.6. Эволюция производительности

Переход от механических и электромеханических устройств к электронным устройствам на вакуумных лампах привел к сто- и даже тысячекратному увеличению скорости вычислений, которая стала измеряться не в секундах, а в миллисекундах. Когда лампы заменили транзисторами, скорость вычислений увеличилась еще в 1000 раз. Эта технология продолжает развиваться, а одним из последних нововведений в архитектуре компьютеров стало использование кэша и конвейеров, значительно повышающих производительность системы.

1.9. Резюме

В этой главе было затронуто множество аспектов структуры и функционирования компьютеров. Были введены основные термины, необходимые для обсуждения поставленных вопросов, и представлены важнейшие архитектурные концепции. В последующих главах мы рассмотрим все эти концепции в более полном объеме.

Упражнения

1.1. Перечислите шаги, необходимые для выполнения машинной команды

Add LOCA,R0

для компьютера, архитектура которого представлена на рис. 1.2. Предполагается, что сама команда хранится в памяти по адресу INSTR и что этот адрес уже находится в регистре PC. Первые два шага по выполнению команды можно определить так:

- пересылка содержимого регистра PC в регистр MAR;
- выдача в память сигнала Read с последующим ожиданием, пока запрошенное слово не будет переслано в регистр MDR.

Не забудьте о том, что регистр PC необходимо обновить и для выборки следующей команды заменить в нем значение INSTR значением INSTR+1.

1.2. Повторите задачу 1.1 для машинной команды

```
Add  R1,R2,R3
```

обсуждавшейся в разделе 1.6.3.

1.3. а) Приведите краткую последовательность машинных команд, необходимую для выполнения такой задачи: «Прибавить содержимое памяти, находящейся по адресу A, к содержимому памяти, расположенной по адресу B, и поместить результат в память по адресу C». Для пересылки данных между памятью и регистрами общего назначения R*i* могут использоваться только две следующие команды:

```
Load   LOC,Ri
Store  Ri,LOC
```

Команды сложения описаны в разделах 1.3 и 1.6.3. Не уничтожайте содержимое ячеек A и B.

б) Предположим, компьютер поддерживает команды Move и Add формата

```
Move/Add  Location1,Location2
```

Команда Move перемещает копию операнда, расположенного по первому адресу, в память по второму адресу, а команда Add прибавляет копию операнда, расположенного по первому адресу, к содержимому памяти по второму адресу. При этом обе команды перезаписывают содержимое памяти по второму адресу. Адрес Location*i* может указывать либо на некоторое место в основной памяти, либо на регистр процессора. Можно ли выполнить задачу 1.3, *а* с помощью меньшего количества команд? Если да, приведите необходимую для этой цели последовательность команд.

1.4. а) В разделе 1.5 рассматривался процесс параллельного выполнения шагов набора программ (рис. 1.4) с целью сокращения общего времени их реализации. Предположим, что каждый из шести интервалов времени, в течение которых выполняются программы операционной системы, равен 1 единице времени, каждая дисковая операция занимает 3 единицы, а каждый интервал, в течение которого выполняется прикладная программа, — 2 единицы времени. Вычислите отношение между временем параллельного выполнения и временем последовательного выполнения для последовательности программ. Начальные и конечные операции пересылки информации не учитывайте.

б) В разделе 1.5 указывалось, что программа может выполняться одновременно с операциями ввода-вывода. Каким будет соотношение между временем реализации программы в режиме параллельной обработки и в режиме последовательной обработки для набора программ при условии, что каждая программа состоит из равного количества операций, производящих вычисления, ввод и вывод? Учитывать относительно небольшое время, уходящее на выполнение программ операционной системы, не нужно.

1.5. а) Для некоторой программы, написанной на языке высокого уровня, нужно оценить время ее выполнения T, определяемое формулой из раздела 1.6.2. Программа может выполняться как на компьютере RISC, так и на компьютере CISC. В обоих компьютерах применяется конвейерная обработка, но в RISC-машине она реализована эффективнее. В частности, эффективное значение параметра S в формуле для вычисления T машины RISC равно 1,2, а для машины CISC — 1,5. У обеих машин одна и та же тактовая частота R. Каково наибольшее допустимое значение параметра N, представляющего количество команд, которые реализуются на CISC-компьютере, выраженное в процентах от значения параметра N для RISC-компьютера, если время выполнения программы на CISC-машине не должно превышать времени выполнения программы на RISC-компьютере?

б) Повторите предыдущую задачу, но при условии, что тактовая частота R RISC-машины на 15 % выше тактовой частоты CISC-компьютера.

1.6. а) Обратимся к кэшу процессора, схематически показанному на рис. 1.5 и описанному в разделе 1.6. Предположим, что время выполнения программы прямо пропорционально времени доступа к команде и что время доступа к команде в кэше в 20 раз меньше времени доступа к команде в основной памяти. Допустим также, что вероятность наличия запрошенной команды в кэше составляет 0,96, но в том случае, если она там не будет обнаружена, то должна сначала быть считана из памяти в кэш, а затем выбрана из кэша и выполнена. Вычислите соотношение времени выполнения программы без использования кэша и времени выполнения программы с использованием кэша. Это соотношение часто называют коэффициентом ускорения, обеспечиваемого наличием кэша.

б) Предположим, что с увеличением вдвое размера кэша в столько же раз уменьшается вероятность отсутствия в нем команды. Повторите предыдущую задачу при условии, что размер кэша увеличен.

Глава 2

Машинные команды и программы

- Машинные команды, выполнение программ и подпрограмм
- Представление чисел, сложение и вычитание в системе дополнения до двух
- Способы адресации для доступа к регистрам и памяти
- Язык ассемблера для представления машинных команд, данных и программ
- Управляемые программой операции ввода-вывода
- Операции со стеком, очередью, списками, связными списками и массивами

В настоящей главе процесс выполнения программы компьютером рассматривается с точки зрения машинных команд. Вы уже знакомы с общей концепцией хранения в памяти команд программы и обрабатываемых ею данных. Теперь речь пойдет о пересылке последовательности команд программы из памяти в процессор и их выполнении. Вы познакомитесь со способами адресации, используемыми для доступа к операндам в памяти и регистрах процессора.

Обо всем этом мы поговорим на уровне базовых концепций. Будет приведено общее описание машинных команд и методов адресации операндов, типичных для коммерческих процессоров. Рассматриваемых в этой главе команд вполне достаточно для создания полноценных реальных программ, выполняющих простые задачи. Для написания таких программ используется не машинный язык, а язык ассемблера, в котором команды и информация об адресах операндов представляются посредством символических имен. Полный набор команд, которые может выполнять данный процессор, часто называют *архитектурой системы команд процессора* (Instruction Set Architecture, ISA) или просто *системой команд*. Система команд определяет не только сами команды, но еще и методы адресации, используемые для доступа к операндам в памяти и регистрах процессора. Для того чтобы рассмотреть базовые концепции, представленные в этой главе, не обязательно изучать всю систему команд компьютера, поэтому здесь приведены лишь наиболее часто используемые. Для иллюстрации излагаемых положений дается много примеров.

В главе 3 представлены системы команд трех коммерческих процессоров, производимых компаниями ARM, Motorola и Intel. Примеры программ из этой главы, написанные на универсальном языке ассемблера (условном языке, содержащем типичные команды ассемблера, но не являющимся набором команд какого-нибудь конкретного процессора), в главе 3 будут повторены для трех конкретных процессоров с использованием их реальных систем команд.

Подавляющее большинство программ пишется на языках высокого уровня, таких как C, C++, Java и FORTRAN. Для этой книги мы выбрали язык ассемблера, который наилучшим образом позволяет показать, как работает компьютер. Компьютер сможет выполнить программу, написанную на языке высокого уровня, лишь при условии, что эта программа сначала будет переведена на машинный язык. Языки высокого уровня очень далеки от машинного языка, и их команды часто транслируются в длинные последовательности машинных команд. Иное дело язык ассемблера. Фактически это просто читабельная форма машинного языка. Так что, изучая его, вы изучаете команды машинного языка, непосредственно выполняемые процессором. Взаимосвязь между языками высокого уровня и машинным языком обязательно должна учитываться при разработке компьютера. И мы с вами еще не раз вернемся к этому вопросу.

Все компьютеры работают с числами. У них имеются команды для осуществления базовых арифметических операций с данными. Кроме того, при выполнении машинных команд программы выполняется ряд арифметических операций, генерирующих числовые адреса для доступа к хранящимся в памяти операндам. Для того чтобы понять, как решаются эти задачи, читатель должен знать, как числа представлены в компьютере и каким образом они складываются и вычитаются. Именно с этого вопроса мы и начнем изложение материала данной главы. Подробное описание логических схем, реализующих компьютерную арифметику, приведено в главе 6.

Компьютеры работают не только с числовыми данными, но и с символами и даже со строками символов, то есть наряду с числовой информацией оперируют и текстовой. Поэтому наряду с числами и операциями над ними будет рассмотрено машинное представление символов.

2.1. Числа, арифметические операции и символы

Компьютеры состоят из логических схем, которые обрабатывают информацию в виде электрических сигналов, принимающих два значения (см. приложение А). Мы обозначаем их цифрами 0 и 1. Количество информации, представленной таким сигналом, измеряется в *битах*. (Слово «бит» произошло от английского словосочетания binary digit, что переводится как «двоичная цифра».) Наиболее естественный способ представления числа в компьютерной системе заключается в использовании строки битов, называемой двоичным числом. Символ текста тоже может быть представлен строкой битов, называемой кодом символа.

Для начала мы опишем представление чисел и арифметические операции над ними в двоичной системе счисления, а затем поговорим о представлении символов.

2.1.1. Представление чисел

Рассмотрим n-разрядный вектор

$$B = b_{n-1}...b_1b_0$$

Здесь $b_i = 0$ или 1 при $0 \le i \le n-1$. Этот вектор может представлять беззнаковое целочисленное значение V в диапазоне от 0 до 2^n-1, где

$$V(B) = b_{n-1} \times 2^{n-1} + ...+b_1 \times 2^1 + b_0 \times 2^0$$

Совершенно очевидно, что нам необходимо как-то представлять и положительные, и отрицательные числа. Существуют три системы представления чисел со знаком:

- значение со знаком;
- дополнение до единицы;
- дополнение до двух.

Во всех трех системах крайний слева бит, называемый *самым старшим разрядом* (Most Significant Bit, MSB), равен 0 в случае положительных чисел и 1 — в случае отрицательных. На рис. 2.1 все три представления показаны на примере 4-разрядных (4-битовых) чисел. Положительные значения во всех трех системах представляются одинаково, а отрицательные — по-разному. В системе значения со знаком отрицательные числа отличаются от соответствующих положительных чисел тем, что значение самого старшего бита (b_3 на рисунке) в векторе B равняется не 0, а 1. Например, число +5 представляется как 0101, а число −5 как 1101. В представлении дополнения до единицы отрицательные значения получают путем дополнения каждого разряда соответствующего положительного значения до единицы. Таким образом, представление числа −3 формируется путем дополнения каждого бита вектора 0011, так что в результате получается 1100. Очевидно, что эту же операцию необходимо выполнить для преобразования отрицательного числа в соответствующее положительное. И в одном и в другом случае преобразование называется дополнением числа до единицы. Операция формирования дополнения заданного числа до единицы эквивалентна вычитанию этого числа из 2^n-1, то есть из 1111 в случае 4-разрядных чисел (см. рис. 2.1). В системе дополнения до двух операция дополнения производится путем вычитания числа из 2^n. То же самое значение можно получить и путем добавления 1 к дополнению этого числа до единицы.

Обратите внимание, что в системах значения со знаком и дополнения до единицы числа +0 и −0 представляются по-разному, а в системе дополнения до двух — одинаково. Имея всего четыре разряда, значение −8 можно представить в системе дополнения до двух, но нельзя представить ни в одной из двух других систем. Для нас наиболее естественной представляется система значения со знаком, поскольку мы привыкли иметь дело с десятичными значениями со знаком. Систему дополнения до единицы относительно легко связать с системой значения со знаком, а вот система дополнения до двух кажется несколько неестественной. Но, как будет показано в разделе 2.1.3, именно она оказалась наиболее эффективным

способом представления чисел с точки зрения выполнения операций сложения и вычитания. Поэтому она чаще всего используется и в компьютерах.

Двоичное значение	Представление числа в системе		
$b_3b_2b_1b_0$	значения со знаком	дополнения до единицы	дополнения до двух
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Рис. 2.1. Двоичное представление целых чисел со знаком

2.1.2. Сложение положительных чисел

Рассмотрим принцип сложения двух одноразрядных чисел. Результат выполнения этой операции приведен на рис. 2.2. Обратите внимание, что для записи результата сложения двух единиц необходим 2-битовый вектор 10, представляющий значение 2. В этом случае говорят, что *сумма* равняется 0, а *перенос* — 1. Для сложения многоразрядных чисел используется метод, аналогичный тому, с помощью которого мы складываем десятичные числа на бумаге. Складываются пары разрядов, начиная с младшего разряда, то есть с правого края битового вектора, с переносом в направлении старшего разряда, то есть левого края битового вектора.

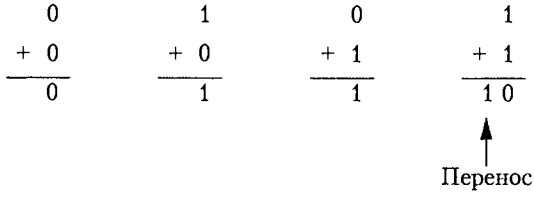


Рис. 2.2. Сложение одноразрядных чисел

2.1.3. Сложение и вычитание чисел со знаком

Итак, вы уже знакомы с тремя системами представления положительных и отрицательных чисел, или, проще говоря, чисел со знаком. Эти системы различаются только способами представления отрицательных значений. Их сравнительные преимущества с точки зрения выполнения арифметических операций можно определить так: простейшая с точки зрения представления чисел система значения со знаком наименее удобна для их сложения и вычитания. Система дополнения до единицы несколько лучше. А наиболее эффективной с точки зрения выполнения указанных операций является система дополнения до двух.

Чтобы понять принципы арифметики дополнений до двух, нужно рассмотреть операцию сложения по модулю N (обозначаемую как mod N). Удобным графическим представлением сложения положительных чисел по модулю N является круг с N значениями по его периметру: от 0 до $N - 1$ (рис. 2.3, *а*). Для примера рассмотрим значение $N = 16$. Результатом операции (7 + 4) mod 16 является значение 11. Для того чтобы выполнить эту операцию с помощью графического представления, найдите на окружности отметку 7 и переместитесь от нее на четыре деления по часовой стрелке. Там вы найдете ответ — значение 11. Аналогичным образом (9 + 14) mod 16 = 7. Найдя значение 9 и отсчитав от него 14 делений, вы опишете полный круг и остановитесь на делении 7. Этот нехитрый графический прием позволяет вычислить любую сумму $(a + b)$ mod 16 для любых положительных чисел a и b: вы находите число a и перемещаетесь на b делений по часовой стрелке.

Теперь рассмотрим другую интерпретацию окружности mod 16. Предположим, что значения из диапазона от 0 до 15 представлены в соответствии с двоичной системой счисления 4-битовыми двоичными векторами: 0000, 0001, ..., 1111. А двоичные векторы, как видно на рис. 2.3, *б*, представляют числа со знаком от −8 до +7, что соответствует системе дополнения до двух (рис. 2.1).

Давайте применим графическую технологию сложения по модулю 16 к простому примеру сложения чисел +7 и −3. В системе дополнения до двух эти числа представлены как 0111 и 1101 соответственно. Для того чтобы их сложить, найдите на окружности число 0111 и переместитесь на 1101 шагов по часовой стрелке (то есть на 13 шагов — сосчитайте количество делений от 0 до 1101). Вы окажетесь на делении 0100, представляющем ответ, а именно +4 (рис. 2.3, *б*). Если вы выполните эту операцию путем сложения пар разрядов справа налево, результат будет таким:

```
  0111
+ 1101
------
1 0100
  ↑
Перенос
```

Как видите, для получения правильного результата мы проигнорировали перенос из четвертого разряда. В этом и состоит суть сложения по модулю. Перемещаясь по кругу (рис. 2.3, *б*), мы возвращаемся не к значению 10000, следующему за значением 1111, а к значению 0000.

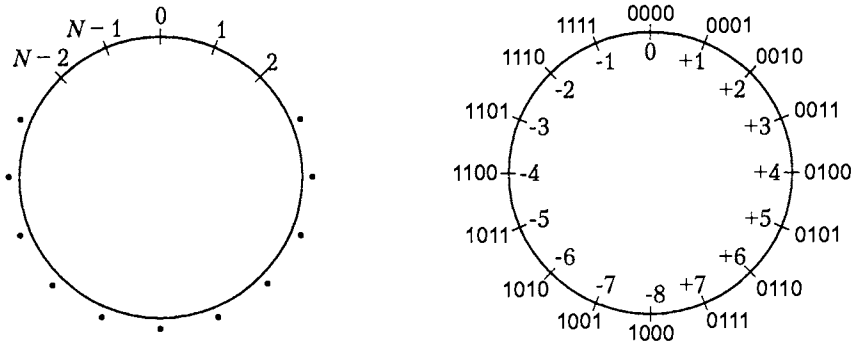


Рис. 2.3. Сложение по модулю и сложение в системе дополнения до двух: представление операций над целыми числами по модулю N (*а*); операции над числами в формате дополнения до двух по модулю 16 (*б*)

Теперь мы можем описать правила сложения и вычитания n-разрядных чисел со знаком в системе дополнения до двух.

1. Для *сложения* двух чисел следует сложить их n-разрядные представления, игнорируя сигнал переноса из позиции старшего разряда (MSB). Суммой будет алгебраически правильное значение, представленное в системе дополнения до двух, если это значение лежит в диапазоне от -2^{n-1} до $+2^{n-1} - 1$.
2. Для *вычитания* чисел X и Y, то есть выполнения операции $X - Y$, следует вычислить дополнение числа Y до двух, а затем добавить его к числу X с учетом правила 1. Результатом будет алгебраически правильное значение, представленное в системе дополнения до двух, если это значение лежит в диапазоне от -2^{n-1} до $+2^{n-1} - 1$.

На рис. 2.4 показано несколько примеров сложения и вычитания 4-разрядных двоичных чисел. Во всех этих примерах ответ оказывается в диапазоне от −8 до 7. Если ответ выходит за границу указанного диапазона, мы говорим, что произошло арифметическое переполнение. Такие ситуации рассматриваются в следующем разделе. Представленные здесь четыре операции сложения (рис. 2.4, *а–г*) выполнены по правилу 1, а шесть операций вычитания (рис. 2.4, *д–к*) — по правилу 2. В операции вычитания для вычитаемого (нижнее значение) сначала выполняется вычисление дополнения, а затем сложение — точно так же, как в случае двух положительных чисел.

В программировании часто возникает необходимость выразить некоторое число, заданное в системе дополнения до двух, с использованием определенного количества разрядов, большего, чем необходимо для представления этого числа на самом деле. Если речь идет о положительных числах, для этого достаточно просто добавить слева нужное количество нулей. В случае отрицательных чисел крайний слева бит, представляющий знак числа, должен быть равен 1, и для получения более длинного представления того же значения нужно повторить знаковый бит слева от числа столько раз, сколько нужно для достижения заданной длины. Чтобы понять, почему нужно действовать именно так, давайте снова вернемся

к окружности для сложения по модулю 16, показанной на рис. 2.3, *б*. Сравните эту окружность с окружностью для сложения по модулям 32 и 64. Представление отрицательных чисел, −1, −2 и т. д., будет точно таким же, с дополнительной единицей слева. Операция добавления единицы называется *расширением знака*.

а	0010	(+2)		*б*	0100	(+4)	
	+ 0011	(+3)			+ 1010	(−6)	
	0101	(+5)			1110	(−2)	
в	1011	(−5)		*г*	0111	(+7)	
	+ 1110	(−2)			+ 1101	(−3)	
	1001	(−7)			0100	(+4)	
д	1101	(−3)			1101		
	− 1001	(−7)	⇒		+ 0111		
					0100	(+4)	
е	0010	(+2)			0010		
	− 0100	(+4)	⇒		+ 1100		
					1110	(−2)	
ж	0110	(+6)			0110		
	− 0011	(+3)	⇒		+ 1101		
					0011	(+3)	
з	1001	(−7)			1001		
	− 1011	(−5)	⇒		+ 0101		
					1110	(−2)	
и	1001	(−7)			1001		
	− 0001	(+1)	⇒		+ 1111		
					1000	(−8)	
к	0010	(+2)			0010		
	− 1101	(−3)	⇒		+ 0011		
					0101	(+5)	

Рис. 2.4. Операции сложения и вычитания в системе дополнения до двух

Теперь вы знаете, насколько просто выполняется сложение и вычитание чисел со знаком в системе дополнения до двух. Поэтому для представления чисел в современных компьютерах выбрана именно эта система. Может показаться, что и система дополнения до 1 не хуже, но это только на первый взгляд. Хотя вычислить дополнение до единицы и проще, результаты операции сложения не всегда оказываются правильными. В данном случае нельзя игнорировать перенос, c_n.

Если $c_n = 0$, полученный результат будет верным. Но если $c_n = 1$, то для определения точного результата к полученному значению нужно добавить 1. Необходимость в этом поправочном цикле, зависящая от значения переноса, делает операции сложения и вычитания в системе дополнения до единицы более сложными, чем в системе дополнения до двух.

2.1.4. Переполнение в целочисленной арифметике

В системе дополнения до двух n бит могут представлять значения из диапазона от -2^{n-1} до $+2^{n-1}-1$. Используя, предположим, четыре бита, можно представить числа от −8 до +7 (рис. 2.1). Когда результат арифметической операции выходит за пределы представимого диапазона, происходит *арифметическое переполнение*. При сложении беззнаковых чисел индикатором переполнения служит перенос c_n из позиции старшего разряда. Однако при сложении чисел со знаком это не срабатывает. Возьмем, к примеру, 4-битовые числа со знаком. Если попытаться сложить числа +7 и +4, результирующим вектором суммы S будет 1011, а это код числа −5. Как видите, результат сложения неверный, хотя сигнал переноса из позиции MSB равен 0. Точно так же при сложении чисел −4 и −6 получим вектор $S = 0110 = +6$, то есть еще один ошибочный результат. Сигнал переноса в данном случае равен 1. Таким образом, переполнение может произойти при условии, что оба слагаемых имеют одинаковый знак. Сложение же чисел с разными знаками не вызывает переполнения. Отсюда можно сделать следующие выводы.

1. Переполнение может произойти только при сложении чисел с одинаковыми знаками.
2. При сложении чисел со знаком сигнал переноса из позиции знакового бита не является индикатором переполнения.

Способ обнаружения переполнения заключается в анализе знаков слагаемых X и Y и знака результата. Если оба операнда, X и Y, имеют один и тот же знак, индикатором переполнения является несовпадение их знака со знаком суммы.

2.1.5. Символы

Компьютеры должны обрабатывать не только числа, но и текстовую информацию, состоящую из символов. Под термином «символы» подразумеваются буквы алфавита, десятичные цифры, знаки препинания и т. п. Они представляются кодами, обычно имеющими длину 8 бит. Одной из наиболее широко распространенных кодовых таблиц является таблица ASCII (American Standard Code For Information Interchange), приведенная в приложении Д.

2.2. Память и адреса

Числовые и символьные операнды, равно как и команды, хранятся в памяти компьютера. Память состоит из многих миллионов *ячеек*, в каждой из которых содержится один бит информации, имеющий значение 0 или 1. Поскольку один бит способен представить очень маленькое количество информации, биты редко обрабатываются поодиночке. Как правило, их обрабатывают группами фиксированного

размера. Для этого память организуется таким образом, что группы по n бит могут записываться и считываться за одну базовую операцию. Группа из n бит называется *словом* информации, а значение n — *длиной слова*. Схематически память компьютера можно представить в виде набора слов (рис. 2.5).

Длина слова современных компьютеров составляет от 16 до 64 бит. Если длина слова компьютера равна 32 битам, в одном слове может храниться 32-разрядное число в системе дополнения до двух или четыре символа ASCII, занимающих по 8 бит (рис. 2.6). Восемь идущих подряд битов называются *байтом*. Для представления машинной команды требуется одно или несколько слов. О кодировании машинных команд мы поговорим далее в этой главе, после того как обсудим команды на уровне языка ассемблера.

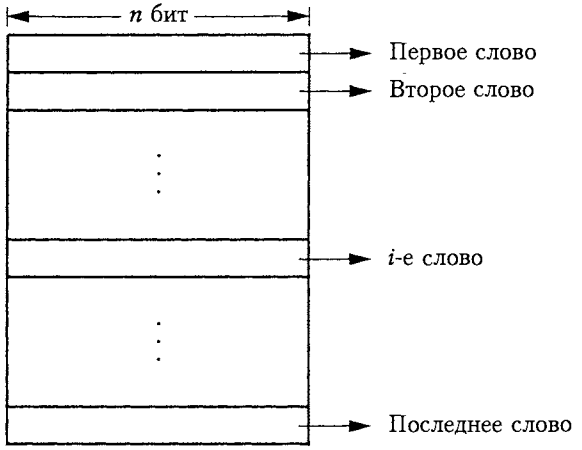
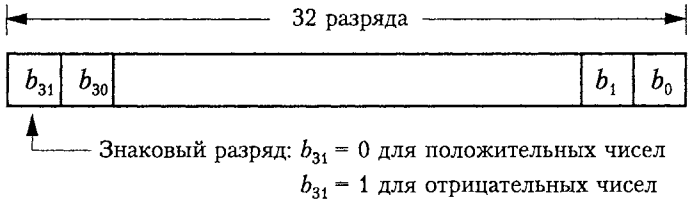
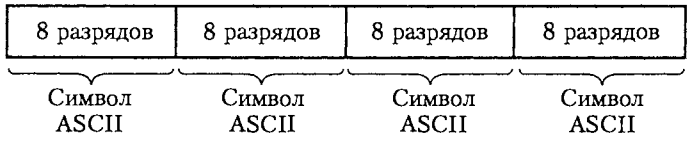


Рис. 2.5. Слова памяти



а



б

Рис. 2.6. Примеры закодированной информации в 32-разрядном слове: целое число со знаком (*а*); четыре символа (*б*)

Для доступа к памяти с целью записи или чтения отдельных элементов информации, будь то слова или байты, необходимы имена или *адреса*, определяющие их расположение в памяти. В качестве адресов традиционно используются числа из диапазона от 0 до 2^k-1 со значением k, достаточным для адресации всей памяти компьютера. Все 2^k адресов составляют *адресное пространство* компьютера. Следовательно, память состоит из 2^k адресуемых элементов. Например, использование 24-разрядных адресов позволяет адресовать 2^{24} (16777216) элементов памяти. Обычно это количество адресуемых элементов обозначается как 16 М (16 мега), где 1 М = 2^{20} (1048576). 32-разрядным адресам соответствует адресное пространство из 2^{32}, или 4 Г (4 гига), элементов, где 1 Г = 2^{30}. Кроме того, часто используются обозначения К (кило), соответствующее 2^{10} (1024), и Т (тера), соответствующее 2^{40}.

2.2.1. Байтовая адресация

Итак, у нас есть три основные единицы информации: бит, байт и слово. Байт всегда равен 8 битам, а длина слова обычно колеблется от 16 до 64 бит. Отдельные биты, как правило, не адресуются. Чаще всего адреса назначаются байтам памяти. Именно так адресуется память большинства современных компьютеров, и именно этот способ адресации мы будем использовать в этой книге. Память, в которой каждый байт имеет отдельный адрес, называется *памятью с байтовой адресацией*. Последовательные байты имеют адреса 0, 1, 2 и т. д. Таким образом, при использовании слов длиной 32 бита последовательные слова имеют адреса 1, 4, 8, ..., и каждое слово состоит из 4 байт.

2.2.2. Прямой и обратный порядок байтов

Существует два способа адресации байтов в словах, а именно в прямом и обратном порядке (рис. 2.7). *Обратным порядком байтов* (big-endian) называется система адресации, при которой байты адресуются слева направо, так что самый старший байт слова (расположенный с левого края) имеет наименьший адрес. *Прямым порядком байтов* (little-endian) называется противоположная система адресации, при которой байты адресуются справа налево, так что наименьший адрес имеет самый младший байт слова (расположенный с правого края). Слова «старший» и «младший» определяют вес бита, то есть степень двойки, соответствующей данному биту, когда слово представляет число (см. раздел 2.1.1). В машинах для коммерческих расчетов используются обе системы адресации. В обеих этих системах адреса байтов 0, 4, 8 и т. д. применяются в качестве адресов последовательных слов памяти в операциях чтения и записи слов.

Наряду с порядком байтов в слове важно также определить порядок битов в байте. Типичный способ расположения битов показан на рис. 2.6, *а*. Это наиболее естественный порядок битов для кодирования числовых данных, непосредственно соответствующий их разрядам. Этот же порядок использован на рисунке при обозначении битов: b_7, b_6, ...,b_0 (слева направо). Однако существуют компьютеры, для которых характерен обратный порядок битов.

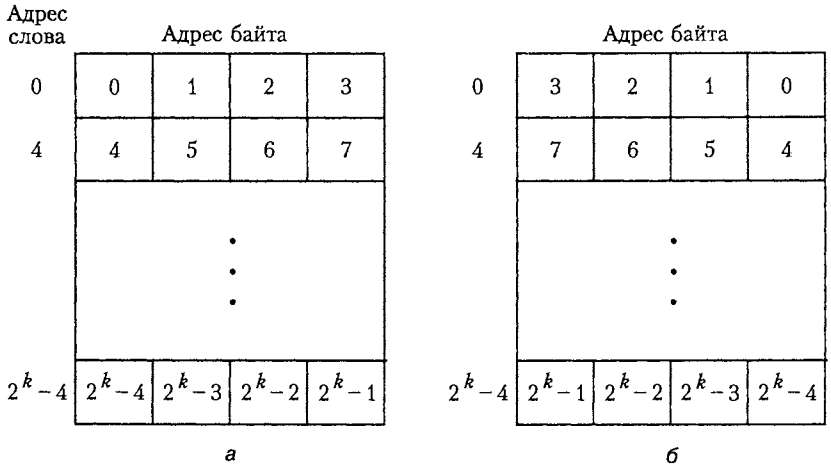


Рис. 2.7. Адресация байтов и слов: обратный порядок байтов (*а*); прямой порядок байтов (*б*)

2.2.3. Расположение слов в памяти

В случае 32-разрядных слов их естественные границы располагаются по адресам 0, 4, 8 и т. д. (рис. 2.7). При этом мы говорим, что слова выровнены по адресам в памяти. Если говорить в общем, слова считаются выровненными в памяти в том случае, если адрес начала каждого слова кратен количеству байтов в нем. По практическим причинам, связанным с манипулированием двоично-кодированными адресами, количество байтов в слове обычно является степенью двойки. Поэтому, если длина слова равна 16 (2 байтам), выровненные слова начинаются по байтовым адресам 0, 2, 4, ..., а если она равна 64 (2^3 байтам), то выровненные слова начинаются по байтовым адресам 0, 8, 16, ...

Не существует причины, по которой слова не могли бы начинаться с произвольных адресов. Такие слова называются невыровненными. Как правило, слова выравниваются по адресам памяти, но в некоторых компьютерах это не так.

2.2.4. Доступ к числам, символам и символьным строкам

Обычно число занимает целое слово. Поэтому, для того чтобы обратиться к нему в памяти, нужно указать адрес слова, по которому оно, это число, хранится. Точно так же доступ к отдельно хранящемуся в памяти символу осуществляется по адресу его байта.

Во многих приложениях необходимо обрабатывать строки символов переменной длины. Для доступа к такой строке нужно указать адрес байта, в котором хранится ее первый символ. Последовательные символы строки содержатся в последовательных байтах. Существует два способа определения длины строки. Первый из них заключается в использовании специального управляющего символа, обозначающего конец строки и являющегося ее последним символом. Второй способ состоит в использовании отдельного слова памяти или регистра процессора, содержащего число, которое определяет длину строки в байтах.

2.3. Операции с памятью

И команды программ, и данные, являющиеся операндами этих команд, хранятся в памяти. Для выполнения команды управляющие схемы процессора должны инициировать пересылку содержащего ее слова или слов из памяти в процессор.

Операнды и результаты также должны пересылаться между памятью и процессором. Таким образом, для выполнения команды программы необходимо произвести две операции с памятью: Load (или Read, или Fetch), то есть *загрузка* (или *чтение*, или *выборка* соответственно) и Store (или Write), то есть *сохранение* (или *запись*).

Операция загрузки пересылает в процессор копию содержимого памяти по заданному адресу. При этом содержимое памяти остается неизменным. Для того чтобы начать операцию загрузки, процессор отсылает в память адрес и запрашивает содержимое памяти по этому адресу. Память считывает соответствующие данные и пересылает их в процессор.

Операция сохранения пересылает элемент информации из процессора в память по заданному адресу, уничтожая предыдущие данные, хранившиеся по этому адресу. Для выполнения такой операции процессор отсылает в память данные и адрес, по которому они должны быть записаны.

Информацию из одного слова или одного байта можно переслать между процессором и памятью за одну операцию. Как рассказывалось в главе 1, процессор содержит небольшое количество регистров, вмещающих по одному слову. Эти регистры служат либо источниками, либо приемниками данных, пересылаемых в память и из памяти. Пересылаемый байт обычно располагается в младшей (крайней справа) позиции в регистре.

Подробности аппаратной реализации указанных операций описываются в главах 5 и 6. В данной же главе мы рассмотрим их с точки зрения системы команд компьютера, то есть сконцентрируем внимание на принципах логической обработки инструкций и операндов. А о конкретных аппаратных компонентах, в том числе о регистрах процессора, мы будем упоминать лишь постольку, поскольку это будет необходимо для понимания процесса выполнения машинных команд и программ.

2.4. Команды и последовательности команд

Задачи, выполняемые компьютерной программой, состоят из последовательности небольших шагов, таких как сложение двух чисел, проверка определенного условия, чтение символа с клавиатуры или вывод такового на экран. Следовательно, компьютер должен поддерживать команды для выполнения следующих четырех типов операций:

- ✦ пересылка данных между памятью и регистрами процессора;
- ✦ арифметические и логические операции с данными;
- ✦ управление последовательностью выполнения программ и их команд;
- ✦ операции ввода-вывода.

Мы начнем со знакомства с командами первых двух типов. Но прежде чем приступать к их анализу, вам следует ознакомиться со специальной нотацией, используемой для описания операций с регистрами.

2.4.1. Нотация для описания операций с регистрами

Нам необходим способ для описания операций пересылки информации из одного места в компьютере в другое. Приемниками и источниками информации могут быть память, регистры процессора и регистры подсистемы ввода-вывода. Как правило, мы будем идентифицировать место хранения информации символическим именем, представляющим его аппаратный двоичный адрес. Например, именами для адресов в памяти могут быть LOC, PLACE, A, VAR2; регистры процессора могут иметь имена R0, R5, а регистры ввода-вывода — имена DATAIN, OUTSTATUS и т. д. Данные, хранящиеся по указанному адресу, обозначаются именем этого адреса, заключенным в квадратные скобки. Таким образом, выражение

$$R1 \leftarrow [LOC]$$

обозначает, что содержимое памяти по адресу LOC пересылается в регистр процессора R1.

В качестве еще одного примера рассмотрим операцию, которая складывает содержимое регистров R1 и R2 и помещает полученный результат в регистр R3. Это действие записывается так:

$$R3 \leftarrow [R1] + [R2]$$

Данный тип записи называется RTN (Register Transfer Notation — нотация регистровых передач). Обратите внимание, что в правой части выражения RTN всегда стоит значение, а в левой — имя того места в памяти, куда его следует поместить, заменив старое содержимое.

2.4.2. Нотация языка ассемблера

Для представления машинных команд и программ нам потребуется другой тип нотации. Для этой цели мы будем использовать формат *языка ассемблера*. Например, команда, выполняющая первую из указанных в предыдущем разделе операций, то есть пересылку данных из памяти в регистр процессора R1 по адресу LOC, записывается так:

```
Move  LOC, R1
```

После выполнения этой команды содержимое памяти по адресу LOC остается неизменным, а старое содержимое регистра R1 перезаписывается.

Во втором примере мы складывали два числа, содержащихся в регистрах процессора R1 и R2, и помещали результат в регистр R3. На языке ассемблера эта операция записывается так:

```
Add  R1,R2,R3
```

2.4.3. Базовые типы команд

Сложение двух чисел относится к числу фундаментальных операций любого компьютера. Инструкция

$$C = A + B$$

в программе на языке высокого уровня — это команда компьютеру сложить текущие значения двух переменных, A и B, и присвоить их сумму третьей переменной, C. При компиляции программы, содержащей эту инструкцию, переменным A, B и C назначаются конкретные адреса памяти. Содержимое памяти по этим адресам представляет значения трех переменных. Поэтому приведенная выше инструкция на языке высокого уровня требует выполнения компьютером следующего действия:

$$C \leftarrow [A] + [B]$$

Для выполнения этого действия содержимое памяти по адресам A и B должно быть переслано в процессор, где будет вычислена сумма. Полученная сумма должна быть отправлена обратно в память и записана по адресу C.

Для начала давайте предположим, что это действие выполняется посредством одной машинной команды. Эта команда содержит адреса трех операндов: A, B и C. Символически такую *трехадресную* команду можно представить как

```
Add  A,B,C
```

Операнды A и B называются *исходными операндами*, а C — *операндом назначения* или *результирующим операндом*. В общем случае команда этого типа имеет такой формат:

```
Операция  Источник1,Источник2,МестоНазначения
```

Если для указания адреса одного операнда в памяти необходимо k бит, в закодированной форме данной инструкции для адресов должно быть отведено $3k$ бит и еще сколько-то бит для кода самой операции Add. В случае современного процессора с 32-разрядным адресным пространством трехадресная команда слишком громоздка для одного слова разумной длины. Поэтому для представления команд такого типа обычно используется формат длиной в несколько слов.

Для выполнения этой же задачи в качестве альтернативы можно использовать несколько более простых команд, с одним-двумя операндами. Предположим, что процессором поддерживаются двухадресные команды в виде:

```
Операция  Источник,МестоНазначения
```

Команда Add такого типа

```
Add  A,B
```

будет выполнять операцию B ← [A] + [B]. После вычисления суммы результат будет переслан обратно в память и сохранен по адресу B с заменой исходных данных,

хранившихся по этому адресу. Это означает, что B является и исходным, и результирующим операндом команды.

Для решения нашей задачи двухадресной команды недостаточно. Потребуется еще одна двухадресная команда, которая копирует значение из одного места памяти в другое. Вот она:

```
Move  B,C
```

Эта команда выполняет операцию C ← [B], оставляя содержимое памяти по адресу B неизменным. Слово Move, означающее «перемещение», использовано здесь не совсем точно: более уместно было бы назвать эту команду Copy (копирование). Однако именно это, первое, название команды давно закрепилось в компьютерном мире. Операция C ← [A] + [B] может быть выполнена с помощью двух команд:

```
Move  B,C
Add   A,C
```

Во всех приведенных выше командах первыми задаются исходные операнды, после них — операнд назначения. Этот порядок характерен для выражений на языке ассемблера, используемых в машинных командах многих компьютеров. Но существует достаточно много компьютеров, в которых порядок операндов обратный. Примеры обоих способов построения команд будут рассмотрены в главе 3. К сожалению, единого соглашения, принятого всеми производителями, не существует. Даже в языке ассемблера одного компьютера могут использоваться команды с разным порядком операндов. В настоящей главе исходные операнды всегда будут задаваться первыми.

Итак, мы определили двух- и трехадресные команды. Но даже двухадресные команды далеко не всегда помещаются в одно слово памяти. Можно, конечно, использовать команды, имеющие по одному операнду. Когда потребуется второй операнд, как в случае команды Add, неявно будет предполагаться, что он находится в конкретном известном месте. Обычно в качестве такого «оговоренного» места служит регистр процессора, называемый *сумматором* (accumulator). Таким образом, *одноадресная* команда

```
Add  A
```

означает следующее: добавить содержимое памяти по адресу A к содержимому сумматора и поместить результат в сумматор. Давайте введем еще две одноадресные команды

```
Load  A
```

и

```
Store  A
```

Команда Load копирует в сумматор содержимое памяти по адресу A, а команда Store копирует содержимое сумматора в память по адресу A. Используя три

одноадресные команды, операцию C ← [A] + [B] можно выполнить следующим образом:

```
Load  A
Add   B
Store C
```

Обратите внимание, что в зависимости от типа команды ее операнд может служить либо источником данных, либо их приемником. Например, в команде Store адрес A определяет приемник данных, а источником является сумматор. А в команде Load, напротив, задается источник данных, а приемником является сумматор.

В некоторых старых компьютерах сумматор был единственным регистром. В современных компьютерах регистров общего назначения обычно довольно много — от 8 до 32, а иногда и больше. Доступ к данным в этих регистрах осуществляется намного быстрее, чем доступ к памяти, поскольку, как уже было сказано, регистры располагаются внутри процессора. А поскольку количество регистров обычно невелико, для того чтобы указать, какой именно регистр участвует в операции, достаточно всего нескольких битов. Так, для задания одного из 32 регистров нужно 5 бит. Это гораздо меньше того количества битов, которое необходимо для задания адреса в памяти. А поскольку регистры позволяют быстрее обрабатывать результаты и применять более короткие команды, они широко используются для временного хранения обрабатываемых данных.

Пусть R*i* — это регистр общего назначения. Команды

```
Load  A,Ri
Store Ri,A
```

и

```
Add   A,Ri
```

представляют собой пример команд Load, Store и Add, в которых R*i* выполняет функцию сумматора. Но даже в том случае, когда в команде явно задается единственный адрес, она все равно может занимать больше одного слова памяти.

Если в процессоре имеется несколько регистров общего назначения, он может поддерживать ряд команд, все операнды которых располагаются в регистрах. Во многих современных процессорах вычисления фактически выполняются непосредственно только над данными, хранящимися в регистрах. К командам такого типа относятся

```
Add   Ri,Rj
```

и

```
Add   Ri,Rj,Rk
```

В обеих этих командах исходными операндами является содержимое регистров R*i* и R*j*. В первой команде регистр R*j* служит и приемником данных, тогда как во второй роль приемника играет третий регистр, R*k*. Такие команды, содержащие лишь имена регистров, обычно умещаются в одно слово.

В программах часто требуется переслать данные из одного места в другое. Для этого используется команда

```
Move  Источник,Приемник
```

помещающая в приемник копию содержимого источника. Команда Move может использоваться для пересылки данных из памяти в регистр процессора и из регистра процессора в память вместо команд Load и Store, поскольку направление пересылки в ней задается просто порядком операндов. Поэтому команда

```
Move  A,Ri
```

означает то же, что и команда

```
Load  A,Ri
```

а команда

```
Move  Ri,A
```

— то же, что и команда

```
Store  Ri,A
```

Мы же в этой главе вместо команд Load и Store будем использовать команду Move.

В тех процессорах, которые производят арифметические операции только над содержимым регистров, задача C = A + B может быть выполнена путем применения такой последовательности команд:

```
Move   A,Ri
Move   B,Rj
Add    Ri,Rj
Move   Rj,C
```

Если процессор позволяет использовать операнды, из которых один находится в памяти, а все остальные в регистрах, то эту же задачу можно реализовать и по-другому:

```
Move   A,Ri
Add    B,Ri
Move   Ri,C
```

Время выполнения конкретной задачи зависит от того, насколько быстро команды пересылаются из памяти в процессор и насколько быстро осуществляется доступ к операндам этих команд. Операции, в которых участвует память, выполняются гораздо медленнее, чем операции с участием регистров. Поэтому значительного ускорения работы можно добиться в тех случаях, когда несколько операций подряд выполняются над хранящимися в регистрах данными без обращения к памяти. При компиляции программ, написанных на языке высокого уровня, в машинный язык важно минимизировать частоту перемещений данных между памятью и регистрами процессора.

Итак, теперь вы имеете представление об одно-, двух- и трехадресных командах и знаете о возможности использовать команды, в которых все операнды задаются неявно. Такие команды применяются в машинах, хранящих операнды в структуре, которая называется *стеком*. А сами команды называются *ноль-адресными*. Концепция стека вводится в разделе 2.8, а компьютер, в котором используется такой подход, описан в главе 11.

2.4.4. Выполнение команд и линейный код

В предыдущем разделе мы рассматривали форматы команд на примере операции C ← [A] + [B]. На рис. 2.8 показан фрагмент программы, выполняющей эту задачу, в том виде, в каком он представлен в памяти компьютера. Мы предполагаем, что компьютер поддерживает только одноадресные команды и что его процессор содержит много регистров. Длина слова составляет 32 разряда, а память адресуется побайтово. Три команды программы расположены в следующих друг за другом словах, начиная с адреса *i*. Поскольку каждая команда занимает 4 байта, вторая и третья команды начинаются по адресам *i* + 4 и *i* + 8. Для того чтобы упростить задачу, мы также предполагаем, что полный адрес целиком задается в команде, занимающей одно слово, хотя при таких размерах адресного пространства и той длине слова, которая поддерживается современными процессорами, подобное, как правило, невозможно.

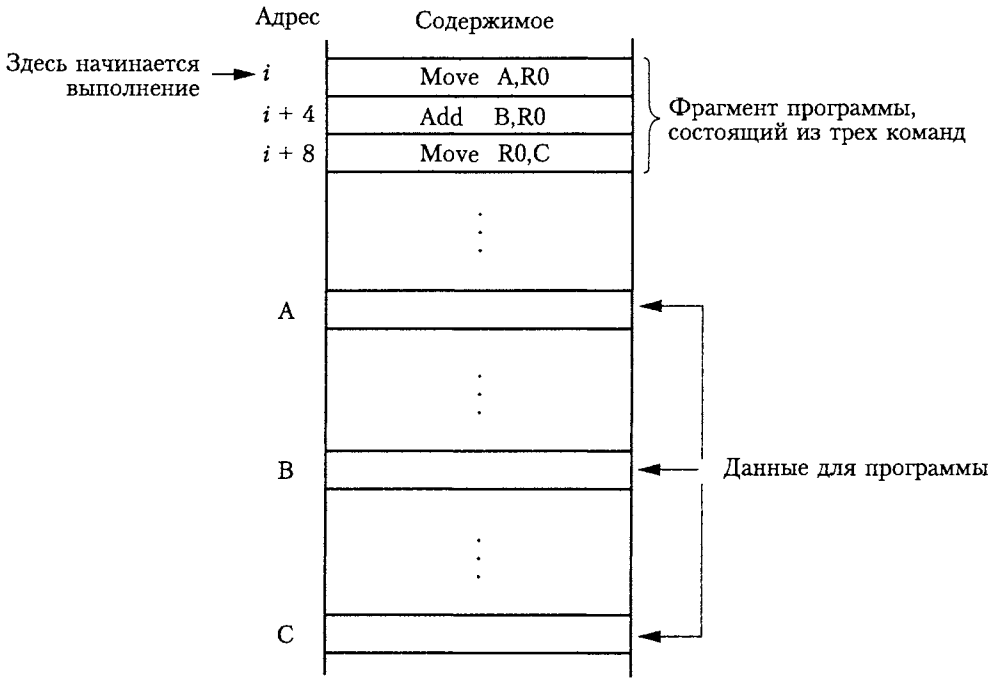


Рис. 2.8. Программа для выполнения операции C ← [A] + [B]

Рассмотрим процесс выполнения этой программы подробно. Процессор содержит регистр, называемый *счетчиком команд* (Program Counter, PC), в котором хранится адрес следующей команды. Чтобы программа начала выполняться, адрес ее первой команды (в нашем примере i) должен быть помещен в регистр PC. Затем управляющие схемы процессора будут использовать информацию из этого регистра для выполнения последовательно расположенных в памяти команд, по одной за раз, в порядке увеличения их адресов. Этот процесс называется *последовательным выполнением* программы, а сама программа, представленная в памяти как список команд, называется *линейной*. В ходе выполнения каждой команды адрес, хранящийся в регистре PC, увеличивается на 4, чтобы этот регистр указывал на следующую команду программы. Таким образом, после выполнения команды Move, расположенной по адресу $i + 8$, регистр PC будет содержать значение $i + 12$, указывающее на первую команду следующего фрагмента программы.

Выполнение команды производится в два этапа. На первом этапе, называемом *фазой выборки* команды, из памяти извлекается команда, хранящаяся по адресу, указанному в регистре PC. Эта команда помещается в *регистр команды* (Instruction Register, IR). После этого начинается второй этап, называемый *фазой выполнения* команды. На этом этапе процессор анализирует команду в регистре IR, чтобы узнать, какое действие ему следует выполнить. Затем он производит это действие, для чего обычно требуется выбрать операнды из памяти или из регистров, выполнить арифметическую или логическую операцию и сохранить результаты по указанному адресу. В ходе этой двухфазной процедуры содержимое регистра PC в определенный момент увеличивается таким образом, чтобы он указывал на следующую команду. Когда выполнение команды завершается, регистр PC содержит адрес следующей команды, то есть можно начинать этап выборки новой команды. В большинстве процессоров фаза выполнения делится, в свою очередь, на несколько фаз, соответствующих выборке операндов, выполнению операции и сохранению результатов.

2.4.5. Ветвление

Ниже будет рассмотрена процедура сложения n чисел. Программа, приведенная на рис. 2.9, является обобщенной версией программы, приведенной на рис. 2.8. Адреса, по которым хранятся n чисел, символически обозначены как NUM1, NUM2, ..., NUMn. Для сложения каждого из этих чисел с содержимым регистра R0 используется отдельная команда Add. После того как все числа сложены, результат записывается в память по адресу SUM.

Но вместо того чтобы использовать длинный список команд Add, можно поместить всего одну такую команду в программный цикл (рис. 2.10). *Цикл* — это последовательность команд, выполняемых необходимое количество раз. В нашей программе цикл начинается по адресу LOOP и заканчивается командой Branch>0. На каждом шаге этого цикла определяется адрес следующего числа, и это число извлекается из памяти и добавляется к содержимому регистра R0. Адрес операнда можно задать разными способами, о которых подробно рассказывается в разделе 2.5. А пока мы должны сосредоточить внимание на том, как создаются и выполняются программные циклы.

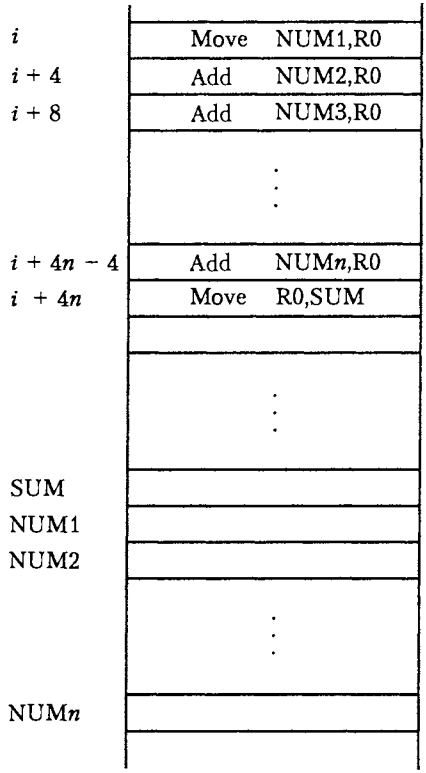


Рис. 2.9. Прямолинейная программа для сложения n чисел

Предположим, что количество складываемых чисел, *n*, хранится в памяти по адресу N (рис. 2.10). Регистр R1 используется в качестве счетчика, определяющего количество повторений цикла. Поэтому в начале программы в регистр R1 загружается содержимое памяти, находящееся по адресу N. В теле цикла команда Decrement R1 уменьшает значение R1 на единицу. (Аналогичная операция выполняется командой Increment, которая увеличивает свой операнд на единицу.) Тело цикла выполняется раз за разом до тех пор, пока результат операции Decrement остается большим нуля.

Теперь нам нужно ввести понятие *команда перехода*. Указанная команда загружает в счетчик команд заданное значение. Процессор выбирает из памяти и выполняет команду, хранящуюся по адресу и определяемую командой перехода, а не следующей командой в порядке увеличения адресов, как в случае прямолинейной программы. Вызов *команды условного перехода* приводит к переходу в другую точку программы только в случае выполнения заданного условия. Если это условие не выполняется, значение в регистре PC увеличивается обычным образом и из памяти выбирается и выполняется очередная команда в порядке увеличения адресов.

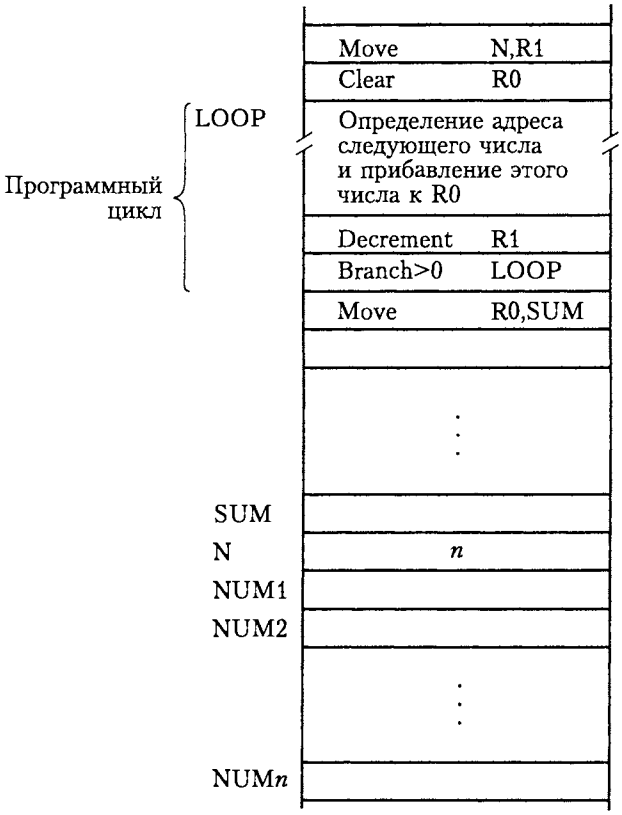


Рис. 2.10. Цикл для сложения n чисел

В программе, представленной на рис. 2.10, команда

Branch>0 LOOP

(«переход, если больше нуля») является командой условного перехода, выполняющей перемещение по адресу LOOP в том случае, если результат предыдущей инструкции, увеличившей значение в регистре R1, больше нуля. Это означает, что цикл повторяется до тех пор, пока в списке чисел остаются необработанные элементы, которые следует добавить к содержимому регистра R0. В конце n-го прохода по циклу команда Decrement возвращает значение 0, поэтому переход на начало цикла не осуществляется, а выполняется следующая по порядку команда — Move. Команда Move перемещает окончательный результат суммирования из регистра R0 в память по адресу SUM.

Проверка условия с последующим выбором одного из нескольких альтернативных путей реализации программы, называемая *ветвлением*, выполняется гораздо чаще, чем просто программные циклы. Такая возможность имеется в системах команд любых компьютеров, поскольку ветвления и циклы относятся к числу фундаментальных операций, без которых невозможно запрограммировать сколько-нибудь нетривиальную задачу.

2.4.6. Флаги кодов условий

Процессор отслеживает результаты выполнения различных операций и сохраняет их для использования в последующих инструкциях условного перехода. Эту информацию он записывает в специальные биты, называемые *флагами кодов условий*. В зависимости от результата выполненной операции отдельные флаги устанавливаются в 1 или 0. Назовем четыре наиболее распространенных флага:

- N (negative) — устанавливается в 1, если результат отрицателен; в противном случае очищается (то есть устанавливается в 0);
- Z (zero) — устанавливается в 1, если результат равен 0; в противном случае очищается;
- V (overflow) — устанавливается в 1, если в результате арифметической операции произошло переполнение; в противном случае очищается;
- C (carry) — устанавливается в 1, если в ходе операции был выполнен перенос; в противном случае очищается.

Флаги N и Z указывают, является результат арифметической операции отрицательным или нулевым. Кроме арифметических команд на эти флаги могут воздействовать команды, выполняющие пересылку данных, такие как Move, Load и Store, благодаря чему ветвление может быть выполнено с учетом знака и значения перемещенного операнда. В некоторых компьютерах имеется также специальная команда Test, анализирующая значение в регистре или в памяти компьютера и устанавливающая либо очищающая флаги N и Z в соответствии с этими значениями. Флаг V указывает на то, что произошло переполнение. Как вы помните, переполнение происходит, когда результат арифметической операции превышает значение, которое можно представить с помощью количества битов, выделенного операнду. Процессор устанавливает флаг V для того, чтобы программист мог определить, что произошло переполнение, и перейти к подпрограмме, способной исправить данную ошибку. Для этой цели используются команды, подобные команде BranchIfOverflow. Кроме того, как будет показано в главе 4, в результате установки бита V может автоматически выполняться программное прерывание, позволяющее решить эту проблему средствами самой операционной системы.

Флаг C устанавливается в 1, если в ходе арифметической операции осуществляется перенос из позиции старшего бита. Этот флаг позволяет выполнять арифметические операции над операндами, длина которых больше длины слова процессора. Такие операции реализуются в арифметике с многократно увеличенной точностью. Более подробно о них рассказывается в главе 6.

Примером команды условного перехода, проверяющей один или более флагов условий, может служить команда Branch>0, описанная в разделе 2.4.5. Она выполняет переход к другой точке программы в том случае, если проверяемое значение не отрицательно и не равно нулю. Это означает, что переход возможен, если ни N, ни Z не равен 1. Существует и множество других команд условного перехода, позволяющих проверять самые разнообразные условия. Задаются такие условия в виде логических выражений, включающих флаги их кодов.

В некоторых компьютерах флаги кодов условий автоматически устанавливаются командами, которые выполняют арифметические и логические операции.

Однако это не всегда так. Многие компьютеры поддерживают две версии команды Add. Одна из них, Add, не воздействует на флаги, а другая, AddSetCC, воздействует. Благодаря этому программисты и компиляторы могут проявлять большую гибкость при создании программ, предназначенных для конвейерного выполнения (см. главу 8).

2.4.7. Формирование адресов памяти

Давайте вернемся к рис. 2.10. Задачей блока команд, составляющих цикл LOOP, является прибавление очередного числа из списка к значению регистра R0 на каждом шаге цикла. Для этого в команде Add, находящейся внутри блока, каждый раз должен указываться новый адрес слагаемого. Как же задается этот адрес? Совершенно очевидно, что задавать его непосредственно невозможно — для этого нам пришлось бы на каждом шаге цикла модифицировать команду Add. Одним из возможных вариантов решения данной проблемы является использование для хранения адреса операнда регистра процессора R*i*. Перед началом цикла в указанный регистр загружается адрес NUM1, и затем на каждом шаге цикла это значение увеличивается на 4.

С описанной ситуацией (и ей подобными) очень часто приходится сталкиваться в программах, следовательно, необходимы гибкие способы адресации операндов. Обычно система команд компьютера поддерживает множество таких способов, называемых *режимами адресации*. Их общая идея одинакова для всех компьютеров, хотя детали могут и различаться. Этой теме посвящен наш следующий раздел.

2.5. Режимы адресации

Итак, мы рассмотрели несколько простых примеров программ на языке ассемблера. Если говорить в общем, каждая программа обрабатывает данные, хранящиеся в памяти компьютера. Эти данные могут быть организованы разными способами. В том случае, если мы будем работать, скажем, с именами студентов, их можно хранить в виде списка. Если же с каждым именем нужно связать дополнительную информацию (например, номера телефонов или оценки по различным предметам), информацию имеет смысл организовать в виде таблицы. Для представления данных, используемых в вычислениях, программисты обычно применяют стандартные способы их организации, называемые *структурами данных*. Примерами структур данных могут служить списки, связные списки, массивы, очереди и т. д.

Программы, как правило, пишутся на языках высокого уровня, позволяющих программистам оперировать константами, глобальными и локальными переменными, указателями и массивами. В процессе трансляции такой программы с языка высокого уровня на язык ассемблера компилятор должен реализовать эти конструкции средствами, предоставляемыми набором команд того компьютера, на котором будет выполняться программа. При этом используются различные способы задания местоположения операндов команд, называемые *режимами адресации*. В данном разделе рассматриваются наиболее важные режимы адресации, поддерживаемые современными процессорами. Их общий список приведен в табл. 2.1.

Таблица 2.1. Стандартные режимы адресации

Адресация	Синтаксис на языке ассемблера	Описание
Непосредственная (immediate)	#Значение	Операнд = Значение
Регистровая (register)	R*i*	EA = R*i*
Абсолютная, прямая (absolute, direct)	LOC	EA = LOC
Косвенная (indirect)	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Индексная (index)	X(R*i*)	EA = [R*i*] + X
Базовая индексная (base, index)	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Базовая индексная со смещением (base, index, offset)	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Относительная (relative)	X(PC)	EA = [PC] + X
Автоинкрементная (autoincrement)	(R*i*)+	EA = [R*i*] Автоувеличение R*i*
Автодекрементная (autodecrement)	−(R*i*)	Автоуменьшение R*i*; EA = [R*i*]

EA — исполнительный адрес (effective address)

Значение — это число со знаком

2.5.1. Реализация переменных и констант

Переменные и константы — это простейшие виды данных, используемые в любой компьютерной программе. Для создания переменной в языке ассемблера резервируется регистр или место памяти, где будет храниться ее значение. В дальнейшем это значение может изменяться с помощью соответствующих команд.

В программах, упоминаемых в разделе 2.4, для доступа к переменным использовались только два режима адресации. Мы обращались к операндам по именам регистров или по их адресам в памяти. Приведем точное определение этих двух режимов адресации.

Регистровая адресация — это режим, в котором операнд является содержимым регистра процессора; в команде задается имя (адрес) регистра.

Абсолютная адресация — это режим, в котором операнд хранится в памяти; его адрес в памяти задается непосредственно в команде. (В некоторых языках ассемблера данный режим адресации называется *прямым*.)

В следующей команде используются оба режима адресации:

```
Move  LOC,R2
```

Если для временного хранения данных задействуются регистры процессора, режим адресации этих данных называется регистровым. Абсолютный режим может использоваться для представления глобальных переменных программы. Например, встретив объявление

```
Integer  A,B;
```

в программе на языке высокого уровня, компилятор выделит адреса памяти для переменных A и B. Теперь, если в программе будут встречаться ссылки на указанные переменные, компилятор сможет генерировать команды на языке ассемблера с абсолютными адресами этих переменных.

Перейдем к константам. Адреса и данные констант могут быть представлены на языке ассемблера с использованием непосредственной адресации. *Непосредственная адресация* — это режим, в котором операнд задается в команде явно. Так, команда

```
Move  200,R0
```

помещает значение 200 в регистр R0. Вы, конечно же, понимаете, что непосредственная адресация может применяться только для задания значения исходного операнда. В языке ассемблера для обозначения непосредственной адресации предназначен не нижний регистр (мы воспользовались им лишь для наглядности), а символ «#» перед значением, применяемым в качестве непосредственного операнда. Вот почему приведенную выше команду следует записать так:

```
Move  #200,R0
```

Значения констант часто используются в программах на языках высокого уровня. Например, инструкция

```
A = B + 6
```

содержит константу 6. Если предположить, что A и B объявлены ранее с применением абсолютного режима адресации, инструкцию можно откомпилировать так:

```
Move  B,R1
Add   #6,R1
Move  R1,A
```

Константы широко используются и в языке ассемблера, где с их помощью выполняются такие операции, как приращение счетчиков, проверка отдельных битов и т. п.

2.5.2. Косвенная адресация и указатели

При использовании описанных в этом разделе режимов адресации операнд и его адрес не задаются прямо в команде. В команде лишь содержится информация, на основании которой можно определить адрес операнда. Этот реальный адрес называется *исполнительным адресом* операнда.

В данном случае речь идет о *косвенной адресации* — режиме, при котором исполнительный адрес операнда находится в регистре или в памяти по адресу, заданному в команде. Для обозначения того, что используется именно косвенная

адресация, задаваемое в команде имя регистра и адрес в памяти заключаются в скобки, как показано на рис. 2.11 и в табл. 2.1.

Для выполнения команды Add, схематически представленной на рис. 2.11, *а*, процессор использует в качестве исполнительного адреса операнда значение B, хранящееся в регистре R1. Он запрашивает из памяти значение, хранящееся по адресу B. Прочитанное значение и является операндом команды, который прибавляется к содержимому регистра R0. Как показано на рис. 2.11, *б*, возможен и другой вариант косвенной адресации, при котором адрес операнда хранится не в регистре, а в памяти. В таком случае процессор сначала считывает содержимое памяти по адресу A, а затем запрашивает вторую операцию чтения, используя в качестве адреса значение B.

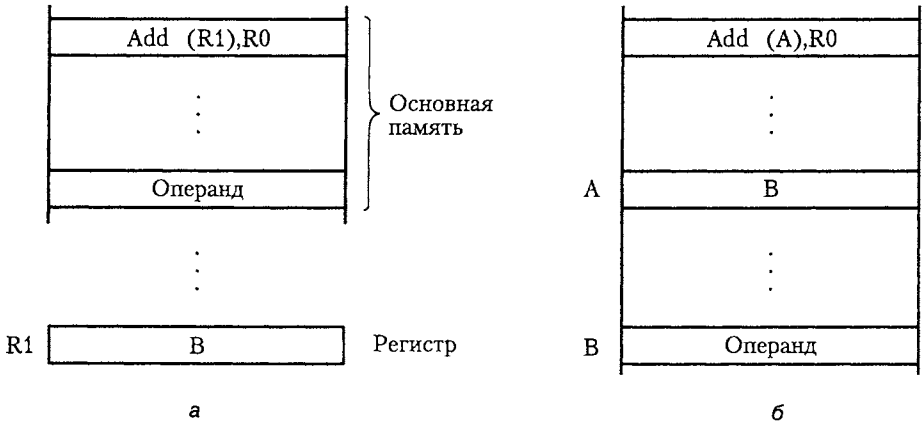


Рис. 2.11. Косвенная адресация: через регистр общего назначения (*а*); через значение в памяти (*б*)

И регистр, и адрес в памяти, содержащий адрес операнда, называются *указателями*. Косвенная адресация и использование указателей являются очень важными и исключительно мощными концепциями программирования. Их суть можно понять, проведя аналогию с процессом поиска сокровищ. Предположим, вам, чтобы найти таковые, велено идти в дом, расположенный по определенному адресу. Но вместо того чтобы обнаружить там сокровище, вы находите записку, в которой указан совершенно другой адрес. Заменив одну записку другой, можно изменить адрес сокровища, но исходное указание (команда) останется неизменным. Изменение содержимого записки эквивалентно изменению содержимого указателя в компьютерной программе. Например, если изменить на рис. 2.11 содержимое регистра R1 или памяти по адресу A, команда Add получит для сложения другой операнд.

Теперь давайте вернемся к программе, складывающей список чисел (рис. 2.10). Для доступа к последовательным числам, хранящимся в этом списке, мы можем применить косвенную адресацию, в результате чего получим программу, представленную на рис. 2.12. Регистр R2 используется в качестве указателя на числа в списке, и доступ к операндам осуществляется косвенно, через этот регистр.

В программе, в разделе инициализации, из памяти по адресу N в регистр R1 загружается значение счетчика *n*. Затем, в режиме прямой адресации, адрес первого элемента в списке, NUM1, помещается в регистр R2. После этого очищается регистр R0 (то есть ему присваивается значение 0). Первые две команды цикла на рис. 2.12 соответствуют не приведенному на рис. 2.10 блоку команд, начинающемуся с метки LOOP. На первом шаге цикла команда

```
Add  (R2),R0
```

извлекает из памяти операнд, хранящийся по адресу NUM1, и прибавляет его к содержимому регистра R0. Вторая команда Add увеличивает значение указателя R2 на 4, чтобы на втором шаге цикла, когда предыдущая команда будет выполняться во второй раз, в нем содержался адрес NUM2.

Адрес	Содержимое		
	Move	N,R1	Инициализация
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Рис. 2.12. Использование косвенной адресации в программе, представленной на рис. 2.10

Рассмотрим инструкцию, написанную на языке C:

```
A = *B;
```

где B — переменная типа указателя. Эту инструкцию можно откомпилировать в следующую пару команд:

```
Move  B,R1
Move  (R1),A
```

Используя косвенную адресацию через память, то же действие можно выполнить с помощью такой команды:

```
Move  (B),A
```

Несмотря на очевидную простоту косвенной адресации через память, этот режим адресации имеет ограниченное применение и в современных компьютерах используется редко. Из главы 8 вы узнаете, что команды, для получения операнда которых приходится дважды обращаться к памяти, плохо совмещаются с режимом конвейерной обработки.

А вот косвенная регистровая адресация применяется весьма широко. Судя по программе, проиллюстрированной на рис. 2.12, этот режим предоставляет разработчикам возможность проявлять определенную гибкость. В тех случаях, когда абсолютная адресация операнда невозможна, косвенная регистровая адресация позволяет обратиться к глобальной переменной, предварительно загрузив ее адрес в регистр.

2.5.3. Индексация и массивы

Режим адресации, который нам сейчас предстоит рассмотреть, также обеспечивает программам большую гибкость, но несколько иного рода. Он полезен в первую очередь для работы с массивами и списками.

Индексная адресация — это режим адресации, при котором исполнительный адрес операнда генерируется путем добавления заданного значения к содержимому регистра.

При индексной адресации можно использовать как специально предназначенный для этого регистр, так и (что бывает чаще) один из регистров общего назначения. В любом случае речь идет об *индексном регистре*. Символическое обозначение индексного режима адресации таково:

$$X(Ri)$$

где X представляет заданное в команде значение константы, а R*i* — имя регистра. Исполнительный адрес операнда этой команды вычисляется так:

$$EA = X + [Ri]$$

В процессе формирования исполнительного адреса операнда содержимое индексного регистра не меняется.

В программе на языке ассемблера константа X может быть задана либо явно в виде числа, либо в виде символического имени, представляющего числовое значение. О том, как устанавливается связь между таким символическим именем и конкретным числовым значением, вы узнаете из раздела 2.6. Когда команда транслируется в машинный код, в нее включается непосредственное значение константы X, обычно представленное меньшим количеством битов, чем слово компьютера. Поскольку константа X является целым числом со знаком, то прежде чем она будет сложена с содержимым регистра, ее знак должен быть расширен до длины этого регистра (см. раздел 2.1.3).

Существует два способа применения индексного режима адресации. На рис. 2.13, *а* индексный регистр R1 содержит адрес в памяти компьютера, а значение X определяет *смещение* операнда относительно этого адреса. Альтернативный способ применения индексной адресации продемонстрирован на рис. 2.13, *б*. Здесь константа X соответствует адресу в памяти, а содержимое индексного регистра определяет смещение операнда относительно данного адреса. В любом из этих двух случаев исполнительный адрес является суммой пары значений, одно из которых явно задается в команде, а другое хранится в регистре.

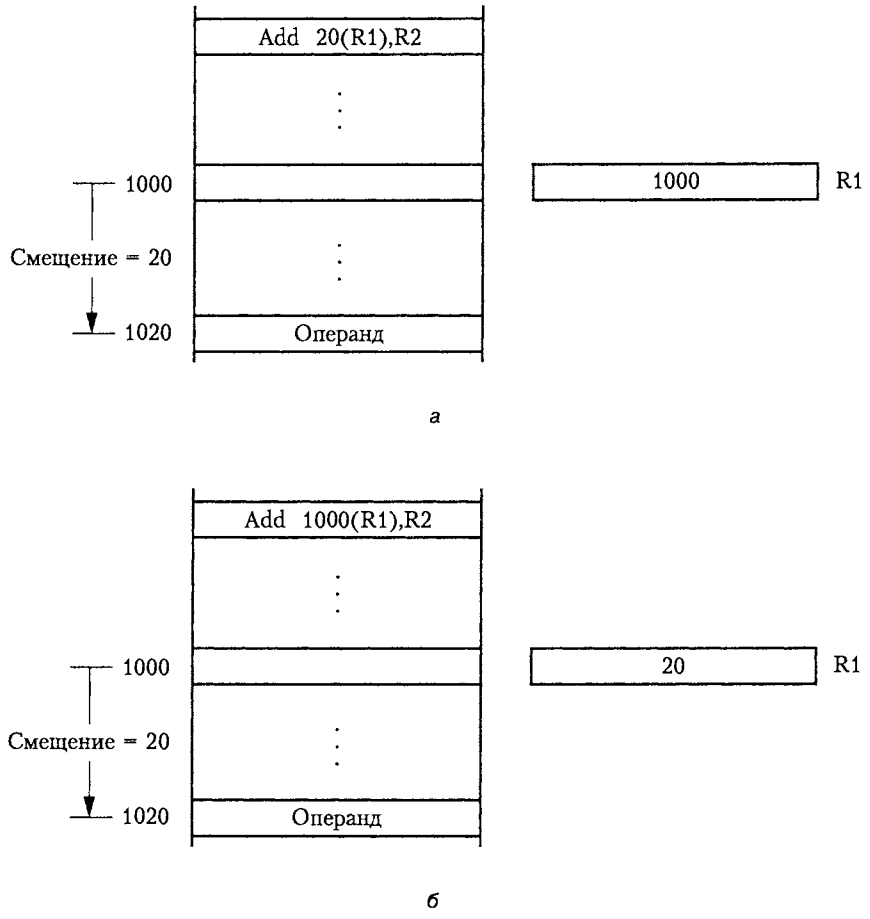


Рис. 2.13. Индексная адресация: смещение задается константой (*а*); смещение задается в индексном регистре (*б*)

Чтобы понять, чем хороша индексная адресация, достаточно рассмотреть простой пример со списком оценок, полученных студентами по некоторому предмету. Предположим, что этот список начинается по адресу LIST и организован так, как показано на рис. 2.14. Информация о каждом из студентов хранится в памяти в виде записи, занимающей блок из четырех слов. Запись состоит из кода студента, за которым следуют оценки, полученные им в результате проведения трех тестов. Всего в группе n студентов, и значение n хранится в памяти по адресу N непосредственно перед списком. Адреса и коды студентов на рисунке указаны исходя из предположения, что память адресуется побайтово и длина слова составляет 32 разряда.

Следует отметить, что список на рис. 2.14 представляет собой двумерный массив, содержащий n строк и четыре столбца. В каждой такой строке содержится запись об одном студенте, а в столбцах указываются коды студентов и их оценки.

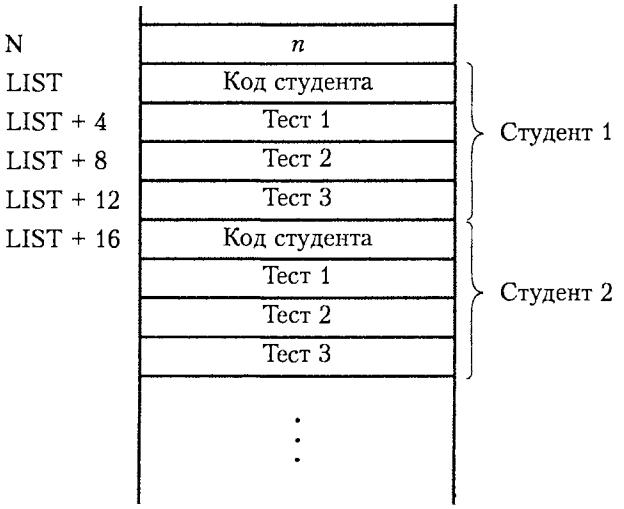


Рис. 2.14. Список оценок студентов

Предположим, мы хотим вычислить суммы баллов, полученных всеми студентами по каждому из трех тестов, и записать эти три суммы в память по адресам SUM1, SUM2 и SUM3. Один из вариантов программы для выполнения такой задачи приведен на рис. 2.15.

```
        Move        #LIST,R0
        Clear       R1
        Clear       R2
        Clear       R3
        Move        N,R4
LOOP    Add         4(R0),R1
        Add         8(R0),R2
        Add         12(R0),R3
        Add         #16,R0
        Decrement   R4
        Branch>0    LOOP
        Move        R1,SUM1
        Move        R2,SUM2
        Move        R3,SUM3
```

Рис. 2.15. Индексная адресация используется для доступа к оценкам студентов в списке, представленном на рис. 2.14

В теле цикла программы для доступа к записям с каждой из трех оценок очередного студента применяется индексный метод адресации, в том виде, в каком

он представлен на рис. 2.13, *а*. В качестве индексного регистра используется R0. Перед началом цикла в этот регистр записывается адрес кода первого студента в памяти компьютера, то есть адрес LIST.

На каждом шаге цикла оценки очередного студента прибавляются к текущим суммам, хранящимся в регистрах R1, R2 и R3 (перед началом цикла все три регистра устанавливаются в 0). Доступ к оценкам осуществляется посредством индексов 4(R0), 8(R0) и 12(R0). Затем значение в индексном регистре увеличивается на 16, чтобы он указывал на код следующего студента. Из содержимого регистра R4, куда перед началом цикла было помещено значение *n*, вычитается 1. На этом очередной шаг цикла заканчивается, а на следующем шаге все повторяется сначала, пока содержимое регистра R4 не станет равным 0. Это будет означать, что оценки всех студентов считаны и цикл должен завершиться. А пока содержимое R4 не равно нулю, с помощью команды условного перехода управление передается обратно на начало цикла, для обработки следующей записи. Последние три команды пересылают итоговые суммы из регистров R1, R2 и R3 в память по адресам SUM1, SUM2 и SUM3.

Следует еще раз подчеркнуть, что содержимое индексного регистра R0 при обращении к записям с оценками студентов не меняется. Оно будет изменено только последней командой Add, предназначенной для перехода от одной записи к другой в конце очередного шага цикла.

Можно сказать, что индексная адресация предназначена для доступа к операнду, расположение которого определено относительно некоторой базовой точки в структуре данных, где хранится этот операнд. В только что рассмотренном нами примере такой базовой точкой служит адрес кода очередной записи о студенте, а оценки по тестам являются операндами, доступ к которым осуществляется в режиме индексной адресации.

Мы обсудили основную форму индексной адресации. Для наиболее эффективного доступа к операндам в памяти в разных ситуациях применяются различные вариации этой базовой формы. Так, смещение X может содержаться во втором регистре. В этом случае для представления операнда используется такая запись:

$$(Ri,Rj)$$

Исполнительный адрес вычисляется как сумма содержимого регистров R*i* и R*j*. Второй регистр обычно называют *базовым*, а содержащееся в нем значение — базой. Данная форма индексной адресации характеризуется большей гибкостью, поскольку позволяет изменять значения обоих компонентов исполнительного адреса.

Чтобы показать, в каких ситуациях может понадобиться такая дополнительная гибкость, мы еще раз обратимся к примеру с данными о студентах, приведенному на рис. 2.14. В программе, представленной на рис. 2.15, в трех командах Add в начале цикла использовались разные значения индекса. Предположим, что каждая запись содержит большое количество элементов — гораздо больше, чем оценок по трем тестам. В таком случае было бы очень хорошо заменить все три команды Add одной командой, выполняемой во втором, вложенном цикле. Подобно тому как для хранения адреса очередной записи используется регистр R0, для хранения смещения очередного элемента записи относительно значения регистра

R0 может быть задействован еще один регистр. На каждом шаге вложенного цикла значение второго регистра будет увеличиваться (см. упражнение 2.9).

Еще одна разновидность индексного режима адресации основывается на использовании двух регистров и константы. Она обозначается так:

$$X(Ri,Rj)$$

В данном случае исполнительный адрес является суммой константы X и содержимого регистров R*i* и R*j*. Этот еще более гибкий способ адресации используется для доступа к нескольким компонентам каждого элемента записи, когда начало элемента записи определяется частью (R*i*,R*j*), а X задает смещение компонента относительного начала элемента. Иными словами, этот режим используется для работы с трехмерными массивами.

2.5.4. Относительная адресация

Итак, мы определили режим индексной адресации операндов с использованием регистров процессора общего назначения. Еще одна интересная разновидность индексного режима адресации связана с использованием вместо регистра общего назначения счетчика команд. Запись X(PC) означает, что исполнительный адрес смещен на X байтов относительно адреса, заданного в счетчике команд. Этот режим называется *относительной адресацией.*

Относительная адресация — это способ адресации, при котором исполнительный адрес определяется так же, как в индексном режиме, но вместо регистра общего назначения используется счетчик команд.

Указанный режим часто используется для доступа к операндам-данным, но более типичным его предназначением является определение целевого адреса в команде перехода. Например, команда

```
Branch>0    LOOP
```

приводит к передаче управления по целевому адресу, определяемому именем LOOP, в том случае, если удовлетворяется условие перехода. Целевой адрес можно не задавать жестко, а вычислять как смещение относительно текущего значения счетчика команд. А поскольку целевой адрес может находиться как выше, так и ниже команды перехода, смещение задается в виде целого числа со знаком.

Вспомните, что в ходе выполнения команды процессор увеличивает значение счетчика, чтобы он указывал на следующую команду программы. В большинстве компьютеров это обновленное значение используется для определения исполнительного адреса в режиме относительной адресации. Предположим, что для задания целевого адреса команды перехода в программе, представленной на рис. 2.12, используется относительная адресация. А еще мы предположим, что четыре команды в теле цикла, начиная с метки LOOP, располагаются в памяти по адресам 1000, 1004, 1008 и 10012. Поэтому на момент, когда генерируется целевой адрес перехода, текущее, уже обновленное, значение в регистре PC равно 1016. Чтобы перейти по адресу LOOP (1000), нужно использовать смещение X = −16.

В языке ассемблера в командах перехода для указания целевого адреса используются метки (рис. 2.12). Когда компилятор при обработке ассемблерной программы встречает такую команду, он вычисляет значение смещения, в данном случае −16, и генерирует соответствующую машинную команду, используя относительную адресацию, то есть −16(PC).

2.5.5. Дополнительные режимы адресации

Вы уже имеете представление о пяти базовых режимах адресации, поддерживаемых большинством компьютеров, а именно о непосредственной, регистровой, абсолютной, косвенной и индексной адресации, а также о нескольких распространенных разновидностях индексного режима, которые используются лишь отдельными компьютерами. И хотя этих режимов вполне достаточно для выполнения любых операций, некоторые компьютеры поддерживают еще и дополнительные режимы, облегчающие решение программных задач. Два таких режима, полезных для доступа к элементам данных, расположенным последовательно в памяти, мы рассмотрим ниже.

Автоинкрементная адресация — это режим адресации, при котором исполнительный адрес содержится в указанном в команде регистре. После обращения к операнду значение в этом регистре автоматически увеличивается таким образом, чтобы он указывал на следующий элемент списка.

В команде на языке ассемблера автоинкрементная адресация обозначается так: имя регистра заключается в скобки, показывая тем самым, что данный регистр содержит исполнительный адрес, а за ним следует знак «+», указывающий, что после обращения к операнду значение в регистре должно быть увеличено:

$$(Ri)+$$

Если режим адресации задается в такой форме, то есть неявно, значит, значение в регистре увеличивается на 1. Однако в случае памяти с побайтовой адресацией такой режим полезен лишь для доступа к последовательным байтам списка. Если же вам требуется получить доступ к последовательным словам, а длина слова составляет 32 разряда, приращение должно быть равным 4. Компьютеры, поддерживающие автоинкрементную адресацию, автоматически увеличивают содержимое регистра на значение, соответствующее размеру операнда. Таким образом, для операнда размером в 1 байт приращение равняется 1, для 16-разрядного операнда — 2, для 32-разрядного операнда — 4. Поскольку размер операнда команды обычно определяется как часть кода операции, в самой команде для указания автоинкрементного режима достаточно обозначения (R*i*)+.

Если компьютером поддерживается автоинкрементный режим, он может использоваться в первой команде Add (рис. 2.12), тогда отпадет надобность во второй команде Add. Измененная подобным образом программа приведена на рис. 2.16.

В дополнение к автоинкрементному режиму адресации компьютером может поддерживаться аналогичный ему режим, предназначенный для доступа к элементам списка в обратном порядке.

Автодекрементная адресация — это режим адресации, при котором содержимое указанного в команде регистра сначала автоматически уменьшается, а затем используется в качестве исполнительного адреса операнда.

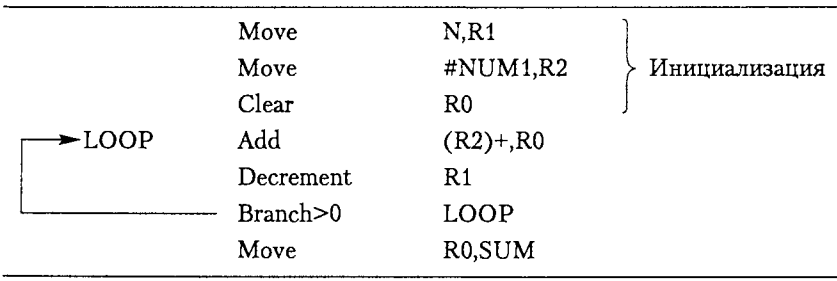


Рис. 2.16. Автоинкрементная адресация в программе, представленной на рис. 2.12

При автодекрементном режиме имя регистра заключается в скобки, перед которыми ставится знак «−», указывающий, что сначала из хранящегося в регистре значения должна быть вычтена некоторая константа:

$$-(Ri)$$

В таком режиме доступ к операндам осуществляется в порядке уменьшения значений адресов. Читатель наверняка заинтересуется, почему адрес уменьшается до использования в команде, а не после, как в случае автоинкрементного режима. Об этом рассказывается в разделе 2.8. Там же мы покажем, как посредством совместного использования этих двух режимов можно реализовать важную структуру данных, называемую стеком.

Операции, выполняемые с помощью автоинкрементного и автодекрементного режимов, можно осуществить и с применением двух команд, одна из которых осуществляет доступ к операнду, а другая увеличивает или уменьшает значение регистра, содержащего адрес операнда. Объединение двух таких операций в одну просто сокращает количество команд, необходимых для выполнения задачи. Однако, как будет показано в главе 8, объединение двух операций в одной команде не всегда является приемлемым решением.

2.6. Язык ассемблера

Машинные команды — это не что иное, как последовательности нулей и единиц. При подготовке программы работать с такими командами, мягко говоря, неудобно. Поэтому для их представления используются символические имена. Как видите, до сих пор для определения команд мы применяли обычные слова английского языка, такие как Move, Add, Increment и Branch, представляющие соответствующие двоичные коды. При написании программ для конкретных компьютеров эти слова обычно заменяются мнемоническими обозначениями, такими как MOV, ADD, INC и BR. Аналогичным образом для ссылки на регистр 3 мы используем обозначение R3, а для ссылки, скажем, на адрес в памяти — обозначение LOC. Полный набор символических имен и правил их применения составляет язык программирования, обычно называемый *языком ассемблера*. Набор правил использования мнемонических обозначений для формирования команд определяет *синтаксис* языка.

Программы, написанные на языке ассемблера, могут автоматически транслироваться в последовательность машинных команд с помощью программы, называемой *ассемблером*. Ассемблер — это одна из служебных программ, входящих в состав системного программного обеспечения. Как и любая другая программа, ассемблер хранится в памяти компьютера в виде последовательности машинных команд. Пользовательская программа обычно вводится в компьютер с помощью клавиатуры и сохраняется либо в памяти, либо на магнитном диске. На этом этапе она представляет собой просто набор строк, состоящих из буквенно-цифровых символов. Для ее обработки запускается ассемблер, который считывает пользовательскую программу, анализирует ее и генерирует соответствующую программу на машинном языке. Эта программа состоит из последовательности нулей и единиц, определяющих команды, которые должны быть выполнены компьютером. Пользовательская программа в исходном текстовом формате называется *исходной*, а ассемблированная программа на машинном языке — *объектной*. О работе ассемблера мы поговорим в разделе 2.6.2, а пока давайте рассмотрим некоторые аспекты языка ассемблера.

В языке ассемблера конкретного компьютера регистр символов может учитываться, а может и не учитываться, то есть заглавные и прописные буквы в нем могут различаться, а могут не различаться. В этой книге для обозначения имен и меток мы будем использовать заглавные буквы, что, как вы понимаете, облегчит чтение команд. Например, команду Move мы запишем так:

```
MOVE  R0,SUM
```

Мнемоническое обозначение MOVE представляет собой *код операции*, то есть двоичную последовательность, соответствующую данной операции. Ассемблер транслирует это обозначение в понятный компьютеру двоичный код операции. За мнемоническим обозначением операции следует как минимум один пробел, а затем информация, определяющая операнды команды. В нашем примере исходным операндом является содержимое регистра R0. Далее идет спецификация результирующего операнда, отделенная от исходного операнда лишь запятой, без разделяющего пробела. Результирующий операнд хранится в памяти, а его двоичный адрес представлен именем SUM.

Поскольку существует несколько режимов адресации, позволяющих задавать расположение операндов, язык ассемблера дает возможность указать, какой из режимов адресации используется для конкретного операнда. Например, числовое значение или имя, явно заданное в приведенной выше команде как SUM, может применяться для обозначения абсолютного режима адресации. Символ «#» обычно означает, что операнд указан непосредственно. Так, следующая команда

```
ADD  #5,R3
```

прибавляет число 5 к содержимому регистра R3 и в этот же регистр помещает результат. Использование символа «#» — не единственный способ обозначения непосредственной адресации. В некоторых языках ассемблера на непосредственную адресацию указывает мнемоническое обозначение самой команды. Например, приведенная выше команда сложения может быть записана так:

```
ADDI  5,R3
```

Суффикс I в мнемоническом обозначении ADDI говорит о том, что исходный операнд команды задан непосредственно (immediate).

Косвенная адресация обычно обозначается с помощью скобок, в которые заключается имя операнда, или с помощью специального символа, обозначающего указатель на операнд. Например, если число 5 должно быть помещено в память по адресу, хранящемуся в регистре R2, соответствующее действие задается так:

```
MOVE  #5,(R2)
```

или так:

```
MOVEI  5,(R2)
```

В данном случае суффикс I указывает на косвенную адресацию (indirect).

2.6.1. Директивы ассемблера

В дополнение к использованию механизма представления команд в программе язык ассемблера позволяет программистам задавать информацию совсем иного рода, необходимую для правильной трансляции исходной программы в объектную. Мы уже упоминали, что всем задействуемым в программе именам при трансляции назначаются числовые значения. Предположим, что имя SUM используется для представления значения 200. Для того чтобы сообщить об этом факте ассемблеру, нужно включить в исходную программу такую инструкцию:

```
SUM  EQU 200
```

Эта инструкция не соответствует команде, которая будет выполнена при реализации объектной программы. Более того, в объектной программе ее вообще не будет. Данная инструкция просто информирует ассемблер, что везде, где в программе встречается имя SUM, оно должно быть заменено значением 200. Такие инструкции, называемые *директивами ассемблера*, используются при трансляции исходной программы в объектную.

Давайте вернемся к программе, схематически представленной на рис. 2.12. Для того чтобы выполнить эту программу на компьютере, нужно написать ее исходный код на соответствующем языке ассемблера, задействовав всю информацию, необходимую для трансляции исходной программы в объектную. Предположим, что каждая команда и каждый элемент данных занимают по одному слову памяти. Это упрощенное представление принято исключительно для данного примера. К тому же мы предположим, что память адресуется побайтово и что длина слова составляет 32 разряда. Объектная программа должна быть загружена в основную память, как показано на рис. 2.17. На этом рисунке указаны адреса памяти, по которым должны располагаться машинные команды и элементы данных после того, как программа будет загружена для выполнения. Если ассемблер должен сгенерировать объектную программу в соответствии с этими условиями, ему необходимо знать:

- как интерпретировать имена;
- куда поместить команды в памяти;
- куда поместить данные-операнды в памяти.

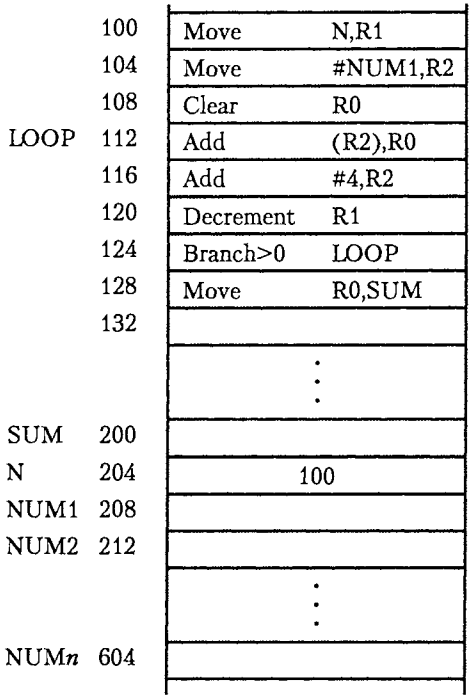


Рис. 2.17. Организация памяти для программы, представленной на рис. 2.12

Чтобы предоставить ассемблеру эту информацию, исходная программа должна быть подобна проиллюстрированной на рис. 2.18. Эта программа начинается с директив ассемблера. Мы уже обсуждали директиву Equate, EQU, информирующую ассемблер о значении константы SUM. Вторая директива, ORIGIN, указывает ассемблеру, куда поместить следующий блок данных. В нашем случае это место памяти определяется адресом 204. По данному адресу должно быть загружено значение 100 (количество элементов в списке), о чем информирует директива DATAWORD.

Для каждой инструкции, в результате выполнения которой в память помещаются команда либо данные, должна быть задана метка. Она будет играть роль идентификатора, которому присваивается адрес соответствующего элемента в памяти. Поскольку директиве DATAWORD назначена метка N, имени N будет присвоено значение 204. Когда это имя будет встречаться в оставшейся части программы, одно будет заменяться данным значением. Такое использование метки N эквивалентно директиве ассемблера

```
N EQU 204
```

Директива RESERVE объявляет, что блок памяти размером 400 байт резервируется для данных, а имя NUM1 будет ассоциироваться с адресом 208. Такая директива не вызывает загрузки по указанному адресу каких-либо данных. Данные могут загружаться в память посредством процедуры ввода, описанной ниже.

Еще одна директива ORIGIN указывает, что команды объектной программы должны загружаться в память по адресу 100. За ней следуют команды исходной программы, написанные в соответствии с мнемоникой и синтаксисом языка ассемблера. Последней инструкцией исходной программы является директива END, указывающая ассемблеру на конец текста исходной программы. Эта директива содержит метку START, идентифицирующую адрес, с которого должно начинаться выполнение программы.

И наконец, инструкция RETURN. Это директива ассемблера, идентифицирующая точку, в которой выполнение программы должно быть прекращено. Она сообщает ассемблеру о необходимости вставить в программу машинную команду, возвращающую управление операционной системе компьютера.

Большинство языков ассемблера требуют, чтобы инструкции исходной программы записывались в форме

Метка Операция Операнд(ы) Комментарий

Эти четыре *поля* обычно разделяются одним или несколькими пробелами либо табуляторами. *Метка* — это необязательное имя, представляющее адрес в памяти, по которому будет загружена команда на машинном языке, сгенерированная на основе данной инструкции. Метки могут представлять и адреса данных в памяти. На рис. 2.18 вы видите пять меток: SUM, N, NUM1, START и LOOP.

	Метка адреса в памяти	Операция	Адресная информация или данные
Директивы ассемблера	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Инструкции, на основе которых генерируются машинные команды	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Директивы ассемблера		RETURN	

Рис. 2.18. Представление программы, приведенной на рис. 2.17, на языке ассемблера

Для каждой команды или директивы ассемблера в поле «Операция» задается мнемоническое обозначение ее кода. В поле «Операнд» содержится адресная информация для доступа к одному или нескольким операндам, что зависит от типа

команды. Поле «Комментарий» ассемблер просто игнорирует — оно предназначено для документирования программы и облегчения ее понимания.

Мы обсудили лишь базовые характеристики языков ассемблера. Основополагающие элементы языка ассемблера у всех компьютеров одинаковы, но несмотря на это языки разных компьютеров имеют различную сложность и отличаются друг от друга многими деталями.

2.6.2. Ассемблирование и выполнение программ

Прежде чем исходную программу, написанную на языке ассемблера, можно будет выполнить, ее необходимо ассемблировать (транслировать) в объектную программу на машинном языке. Эта задача осуществляется программой-ассемблером, заменяющей все символические обозначения операций и адресных режимов двоичными кодами, используемыми в машинных командах, а все имена и метки — их настоящими значениями.

Ассемблер присваивает командам и блокам данных адреса, начиная с адреса, заданного в директиве ORIGIN. Эта директива отвечает за размещение по нужным адресам констант, которые могут быть заданы в директивах DATAWORD, и резервирует память в соответствии с указаниями, содержащимися в директивах RESERVE.

Важнейшим моментом процесса ассемблирования является определение значений, заменяющих имена. В тех случаях, когда значение имени определяется директивой EQU, это очень простая задача. Но если имя задается в поле «Метка» команды, представляемое им значение определяется положением этой команды в ассемблированной объектной программе. Поэтому в ходе формирования машинного кода ассемблер должен отслеживать адреса, по которым будут загружаться последовательные команды программы. Например, именам START и LOOP будут назначены значения 100 и 112 соответственно.

В некоторых случаях ассемблер не заменяет представляющее адрес имя реальным значением этого адреса. В частности, в командах перехода не заменяется реальным адресом имя, определяющее точку, к которой будет выполняться переход. В командах перехода целевой адрес обычно задается в режиме относительной адресации (см. раздел 2.5). Ассемблер вычисляет *смещение перехода*, то есть расстояние до целевого адреса, и помещает его в машинную команду.

Сканируя исходную программу, ассемблер записывает все имена и соответствующие им числовые значения в специальную таблицу, называемую *таблицей символов*. Когда имя встречается в программе повторно, оно заменяется соответствующим значением из таблицы. Проблема может возникнуть лишь в том случае, если имя использовалось в качестве операнда до того, как ему было присвоено значение. Такое возможно, в частности, при выполнении перехода вверх по программе. Ассемблер не сможет определить целевой адрес перехода, так как указанное в команде имя еще не записано в таблицу символов. Для решения проблемы достаточно дважды просканировать исходную программу. На первом проходе ассемблер создает полную таблицу символов. В конце этого прохода всем встречающимся в программе именам должны быть поставлены в соответствие числовые значения.

После этого ассемблер снова проходит по исходной программе и заменяет все имена значениями из таблицы символов. Такой ассемблер называется *двухпроходным*.

Ассемблер сохраняет объектную программу на магнитном диске. Перед выполнением эта программа загружается в память компьютера. Для этого там должна присутствовать еще одна служебная программа, называемая *загрузчиком*. Загрузчик выполняет последовательность операций ввода, необходимых для пересылки программы на машинном языке с диска в заданное место памяти. При этом загрузчик должен знать длину программы и адрес, по которому ее необходимо загрузить. Обычно ассемблер помещает такого рода информацию в заголовок программы, предшествующий объектному коду. Загрузив код, загрузчик начинает выполнение объектной программы с перехода к той ее команде, которая должна быть реализована первой. Адрес этой команды включается в программу на языке ассемблера в качестве операнда директивы ассемблера END. Ассемблер помещает этот адрес в заголовок, предшествующий объектному коду на диске.

Программа начинает выполняться и выполняется до своего завершения, если только в ней нет логических ошибок. Пользователь должен уметь находить такие ошибки. Что касается синтаксических ошибок, то ассемблер выявляет их сам и сообщает об этом пользователю. Для того чтобы помочь пользователю найти другие программные ошибки, в состав системного программного обеспечения обычно включают программу, называемую *отладчиком*. Эта программа предоставляет пользователю возможность остановить выполнение объектной программы в указанной им точке и проанализировать содержимое памяти и регистров процессора. Об отладке программ подробнее рассказывается в главе 4.

2.6.3. Запись чисел

При работе с числовыми значениями обычно используют привычную десятичную запись. Конечно, в компьютере числовые значения хранятся как двоичные числа. В некоторых случаях удобнее прямо задавать двоичные значения. Большинство реализаций ассемблера позволяет задавать числовые значения различными способами, используя соглашения, определяемые синтаксисом языка. Рассмотрим в качестве примера число 93, представленное в виде 8-разрядной двоичной записи 01011101. Если непосредственно использовать это значение в качестве операнда, его можно задать в виде десятичного числа, как в следующей команде:

```
ADD  #93,R1
```

или же в виде двоичного числа, на что указывает специальный префикс, такой как символ «%»:

```
ADD  #%01011101,R1
```

Двоичные числа можно записать более компактно, используя *шестнадцатеричную* систему, в которой каждые четыре бита числа представлены одной шестнадцатеричной цифрой. Шестнадцатеричную запись можно считать расширенной версией двоично-десятичной записи (см. приложение Д). Первые десять кодов, 0000,

0001, ..., 1001, обозначаются цифрами 0, 1, ..., 9, как в двоично-десятичной системе кодирования. Оставшиеся шесть 4-разрядных кодов, 1010, 1011, ..., 1111, обозначаются буквами A, B, ..., F. В шестнадцатеричном представлении десятичное значение 93 записывается как 5D. В языках ассемблера шестнадцатеричное представление часто задается с помощью префикса в виде знака доллара. Таким образом, команда, прибавляющая к содержимому регистра число 93, при использовании шестнадцатеричной системы кодирования чисел записывается так:

```
ADD  #5D,R1
```

2.7. Базовые операции ввода-вывода

В предыдущем разделе описывались машинные команды и режимы адресации. При этом мы предполагали, что данные, на которые воздействуют указанные команды, уже хранятся в памяти. Далее речь пойдет о средствах, с помощью которых данные пересылаются между памятью компьютера и внешним миром. Операции ввода-вывода являются одной из важнейших составляющих работы компьютера, и от того, как они выполняются, в значительной мере зависит его производительность. Эта тема подробно освещается в главе 4. А пока мы рассмотрим несколько базовых концепций ввода-вывода.

Предположим, нам необходимо считать вводимые с клавиатуры символы и вывести их на экран. Простейший способ выполнения подобных задач заключается в использовании метода, который называется *программно управляемым вводом-выводом*. Скорость передачи данных от клавиатуры к компьютеру зависит от того, насколько быстро пользователь может их вводить (а это едва ли больше нескольких символов в секунду!). Скорость передачи данных от компьютера к дисплею гораздо выше. Она определяется пропускной способностью соединения между компьютером и дисплеем, которая обычно составляет несколько тысяч символов в секунду. Однако и это невероятно мало, если сравнивать со скоростью работы процессора, выполняющего миллионы команд в секунду. Разница в быстродействии процессора и устройств ввода-вывода вызывает потребность в механизмах синхронизации процесса передачи данных между ними.

Решение этой проблемы заключается в следующем. Процессор отсылает дисплею первый символ и ждет от него сигнала о том, что символ получен. Затем он отсылает второй символ, снова ждет сигнала и т. д. Точно так же отсылаются процессору данные, вводимые с клавиатуры. Процессор ждет сигнала, означающего, что пользователь нажал одну из клавиш и что ее код помещен в некоторый буферный регистр, связанный с клавиатурой. Получив сигнал, процессор считывает код клавиши.

Как показано на рис 2.19, клавиатура и дисплей являются совершенно независимыми устройствами. Нажатие клавиши на клавиатуре само по себе не вызывает вывода на экран соответствующего символа. Один блок команд в программе ввода-вывода пересылает символы в процессор, а другой выводит таковые на экран. Рассмотрим, как пересылается код символа от клавиатуры процессору. В ответ на нажатие клавиши соответствующий ей код символа сохраняется в 8-разрядном буферном регистре, связанном с клавиатурой. Назовем его DATAIN, как

на рис. 2.19. Для уведомления процессора о наличии нового кода символа в регистре DATAIN специальный флаг состояния SIN устанавливается в 1. Программа отслеживает состояние этого флага, и когда он оказывается установленным в 1, процессор считывает содержимое регистра. После пересылки символа в процессор значение флага SIN автоматически сбрасывается в 0. Когда с клавиатуры вводится второй символ, флаг SIN снова устанавливается в 1 и процесс повторяется.

Вывод символа на экран осуществляется аналогичным образом. Но в этом случае для его пересылки от процессора к монитору используются буферный регистр DATAOUT и флаг SOUT. Если значение SOUT равно 1, значит, дисплей готов к приему символа. Процессор под управлением программы отслеживает содержимое флага SOUT, и когда его значение оказывается равным 1, процессор пересылает код символа в регистр DATAOUT. После этого флаг SOUT сразу же снимается. Когда дисплей готов к приему следующего символа, флаг SOUT опять устанавливается в 1 и процесс повторяется. Буферные регистры DATAIN и DATAOUT совместно с флагами состояния SIN и SOUT являются частью схемы, называемой *интерфейсом устройства*. Такая схема имеется у каждого устройства ввода-вывода. С процессором она соединяется через шину (рис. 2.19).

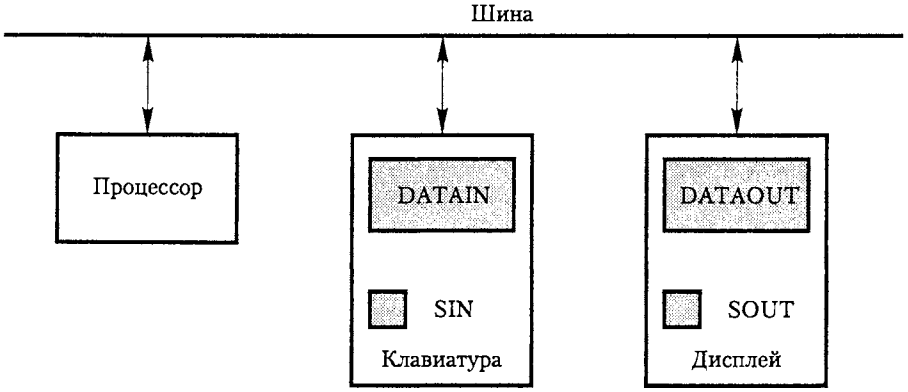


Рис. 2.19. Шина, соединяющая процессор с клавиатурой и дисплеем

Для выполнения операций ввода-вывода необходимы машинные команды, проверяющие состояние флагов и пересылающие данные между процессором и устройствами ввода-вывода. Эти команды сходны с командами пересылки данных между процессором и памятью. Например, процессор может отслеживать состояние флага клавиатуры SIN и пересылать символ из регистра DATAIN в R1 путем применения такой последовательности операций:

```
READWAIT   Branch to READWAIT if SIN = 0
           Input from DATAIN to R1
```

Операция Branch обычно реализуется посредством двух машинных команд. Первая проверяет флаг состояния, а вторая выполняет переход. Процесс отслеживания состояния флага в различных компьютерах может быть реализован немного

по-разному, но основная идея состоит в том, что процессор выполняет короткий *цикл ожидания* и, обнаружив факт установления флага, производит пересылку данных, послед чего флаг снова сбрасывается.

То же самое происходит при выводе данных на дисплей:

```
WRITEWAIT   Branch to WRITEWAIT if SOUT = 0
            Output from R1 to DATAOUT
```

Операция обычно реализуется с помощью двух машинных команд. Цикл ожидания выполняется до тех пор, пока не будет установлен флаг SOUT, означающий, что дисплей готов к приему символа. Закончив пересылку, оператор Output сбрасывает флаг SOUT.

Предполагается, что в исходном состоянии флаг SIN сброшен, а флаг SOUT установлен. Такая инициализация обычно выполняется схемами управления устройством в тот момент, когда устройство переходит под управление компьютера, до начала реализации программы.

До сих пор мы предполагали, что адреса, используемые процессором для доступа к командам и операндам, всегда указывают места в памяти. Но очень часто в компьютерах применяется такая система адресации, когда некоторые адреса указывают на буферные регистры периферийных устройств, такие как DATAIN и DATAOUT. Называется эта система адресации *вводом-выводом с отображением в памяти* (memory-mapped input-output). Таким образом, для получения доступа к регистрам не нужны какие-то особые команды. Данные пересылаются между регистрами и процессором с помощью уже рассмотренных команд Move, Load и Store.

Например, содержимое символьного буфера клавиатуры DATAIN можно переслать в регистр процессора R1 с помощью такой команды:

```
MoveByte  DATAIN,R1
```

Аналогичным способом содержимое регистра R1 можно переслать в регистр DATAOUT:

```
MoveByte  R1,DATAOUT
```

Флаги состояния SIN и SOUT при обращении к регистрам DATAIN и DATAOUT сбрасываются автоматически. Код операции MoveByte означает, что операнд имеет размер 1 байт. Он отличается от кода операции Move, используемого для перемещения операндов длиной в одно слово.

Мы установили, что буферы данных, показанные на рис. 2.19, могут адресоваться так, как если бы они располагались в основной памяти компьютера. Точно так же можно было бы поступить и с флагами SIN и SOUT, присвоив им разные адреса. Но чаще эти флаги включают в состав *регистров состояния устройства* (такой регистр имеется у каждого устройства ввода-вывода). Предположим, что бит b_3 в регистре INSTATUS соответствует флагу SIN, а в регистре OUTSTATUS — флагу SOUT. Следовательно, описанную выше операцию чтения можно выполнить с помощью приведенных ниже команд.

```
READWAIT   TestBit  #3,INSTATUS
           Branch=0 READWAIT
           MoveByte DATAIN,R1
```

А операция записи может быть реализована так:

```
WRITEWAIT  TestBit  #3,OUTSTATUS
           Branch=0 WRITEWAIT
           MoveByte R1,DATAOUT
```

Команда TestBit проверяет состояние одного бита (позиция которого определяется первым операндом) во втором операнде. Если значение этого бита равно 0, значит, условие команды Branch истинно и она выполняет переход на начало цикла ожидания. Когда устройство будет готово (значение проверяемого бита окажется равным 1) данные будут прочитаны из входного буфера или записаны в таковой.

В программе, представленной на рис. 2.20, эти две операции предназначены для чтения введенной с клавиатуры строки символов и вывода ее на экран. Символы поочередно считываются по мере ввода и сохраняются в области данных в памяти, а затем выводятся на экран. Программа завершает свою работу после того, как ею будет прочитан, сохранен и выведен на экран символ возврата каретки CR. Первый байт области памяти, где хранится считываемая с клавиатуры строка, имеет адрес LOC. Этот адрес хранится в регистре R0, куда он записывается первой командой программы. По мере чтения символов значение регистра R0 увеличивается на 1, для чего в команде Compare используется автоинкрементный способ адресации.

	Move	#LOC,R0	Инициализация регистра R0, то есть занесение в него адреса, по которому строка символов будет храниться в памяти
READ	TestBit	#3,INSTATUS	Ожидание ввода символа в буфер клавиатуры DATAIN
	Branch=0	READ	
	MoveByte	DATAIN,(R0)	Пересылка символа из регистра DATAIN в память (эта операция сбрасывает флаг SIN)
ECHO	TestBit	#3,OUTSTATUS	Ожидание готовности дисплея
	Branch=0	ECHO	Пересылка только что прочитанного символа в буферный регистр дисплея (эта операция сбрасывает флаг SOUT)
	MoveByte	(R0),DATAOUT	
	Compare	#CR,(R0)+	Проверка того, является ли только что прочитанный символ символом CR, а также увеличение значения указателя, с тем чтобы он указывал на следующий символ
	Branch≠0	READ	Если считываемый символ не является символом CR, возврат назад и считывание следующего символа

Рис. 2.20. Программа, считывающая строку символов и выводящая ее на экран

Программно управляемый ввод-вывод требует постоянного участия процессора. Практически все время выполнения рассматриваемой программы тратится на два цикла ожидания — процессор просто ждет нажатия клавиши или готовности дисплея. Это, конечно же, не очень хорошо — процессор не должен простаивать. Поэтому для выполнения ввода-вывода применяются другие технологии, основанные на использовании прерываний.

2.8. Стеки и очереди

Компьютерным программам часто приходится выполнять определенные подзадачи с использованием знакомой вам структуры, называемой подпрограммой. Для обмена информацией между главной программой и подпрограммой предназначена специальная структура данных, называемая стеком. В этом разделе рассказывается о стеках и тесно связанной с ними структуре, которая называется очередью.

Обрабатываемые программой данные могут быть организованы множеством способов. Вы уже знакомы с данными, структурированными в виде списков. Теперь мы рассмотрим еще одну важную структуру данных, называемую стеком. *Стек* — это список элементов данных, обычно слов или байтов, доступ к которым ограничен следующим правилом: элементы этого списка могут добавляться только в его конец и удаляться только из конца. Конец списка называется вершиной стека, а его начало — дном. Такую структуру иногда называют *магазином*. Представьте себе стопку подносов в столовой. Клиенты берут подносы сверху, и работники столовой, добавляя чистые подносы, тоже кладут их на верх стопки. Этот механизм хранения хорошо описывается емкой фразой «последним вошел — первым вышел» (Last In First Out, LIFO), означающей, что элемент данных, помещенный в стек последним, удаляется из него первым. Операцию помещения нового элемента в стек часто называют его *проталкиванием* (push), а операцию извлечения последнего элемента из стека называют его *выталкиванием* (pop).

Хранящиеся в памяти компьютера данные могут быть организованы в виде стека, так чтобы последовательные элементы располагались друг за другом. Предположим, что первый элемент хранится по адресу BOTTOM, а когда в стек помещаются новые элементы, они располагаются в порядке уменьшения последовательных адресов. Таким образом, стек растет в направлении уменьшения адресов, что является весьма распространенной практикой.

На рис. 2.21 показано, как располагается в памяти компьютера стек, элементы которого занимают по одному слову. На дне он содержит числовое значение 43, а на вершине — 28. Для отслеживания адреса вершины стека используется регистр процессора, называемый *указателем стека* (Stack Pointer, SP). Это может быть один из регистров общего назначения или же регистр, специально предназначенный для этой цели. Если предположить, что память адресуется побайтово и слово имеет длину 32 разряда, операцию проталкивания в стек можно реализовать так:

```
Subtract    #4,SP
Move        NEWITEM,(SP)
```

где команда Subtract вычитает исходный операнд 4 из результирующего операнда, содержащегося в регистре SP, и помещает результат в регистр SP. Эти две команды помещают слово, хранящееся по адресу NEWITEM, на вершину стека, предварительно уменьшая указатель стека на 4. Операция выталкивания из стека может быть реализована так:

```
Move   (SP),ITEM
Add    #4,SP
```

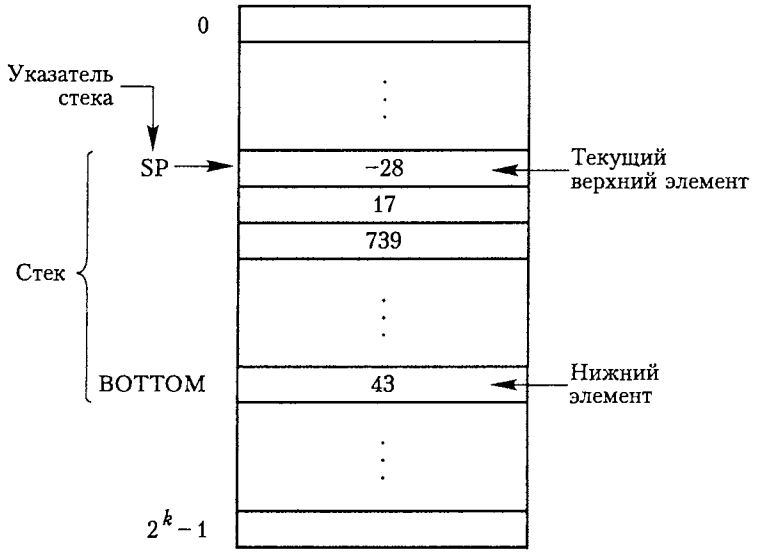


Рис. 2.21. Стек слов в памяти

Эти две команды перемещают значение, хранившееся на вершине стека, в другое место памяти, по адресу ITEM, а затем уменьшают указатель стека на 4, чтобы он указывал на тот элемент, который теперь располагается на вершине стека. Результат выполнения каждой из этих операций для стека, показанного на рис. 2.21, приведен на рис. 2.22.

Если процессор поддерживает режимы автоинкрементной и автодекрементной адресации, для помещения нового элемента в стек достаточно команды

```
Move  NEWITEM,−(SP)
```

А выталкивание элемента из стека можно выполнить посредством команды

```
Move  (SP)+,ITEM
```

Когда стек используется программой, для него обычно выделяется фиксированное количество памяти. В этом случае нам нужно проследить за тем, чтобы программа не пыталась помещать новые элементы в стек, достигший своего максимального размера. Кроме того, она не должна пытаться вытолкнуть элемент из пустого стека, что могло бы произойти в случае логической ошибки.

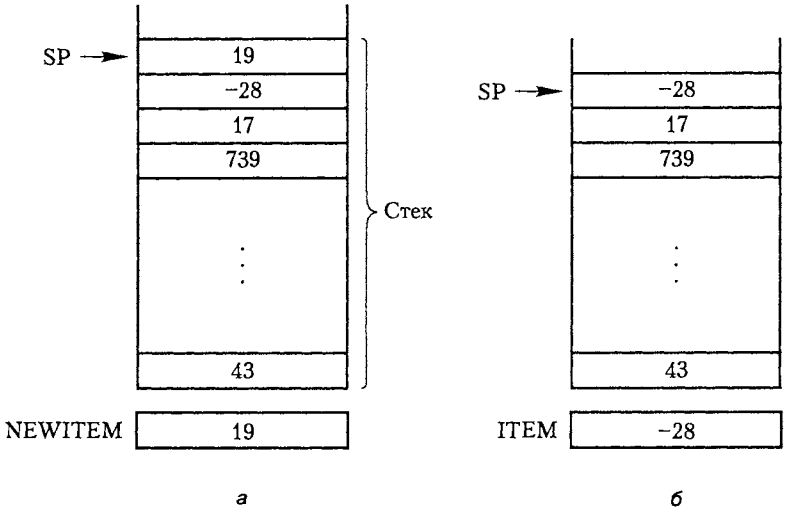


Рис. 2.22. Результат выполнения операций со стеком на примере стека, показанного на рис. 2.21: после помещения в стек элемента NEWITEM (*а*); после выталкивания из стека верхнего элемента и помещения его по адресу ITEM (*б*)

Предположим, что стек заполняется начиная с адреса 2000 (BOTTOM) до 1500 и далее. Первоначально в регистр, играющий роль указателя стека, загружается значение 2004. Напомним, что перед помещением в стек нового элемента данных из значения SP каждый раз вычитается 4. Поэтому начальное значение 2004 означает, что первый элемент стека будет иметь адрес 2000. Для предотвращения попыток помещения элемента в полный стек или удаления элемента из пустого стека нужно несколько усложнить реализацию операций проталкивания и выталкивания элемента. Для выполнения каждой из этих операций указанной выше команды недостаточно — ее нужно заменить последовательностью команд, приведенной на рис. 2.23.

Команда сравнения

```
Compare  src,dst
```

выполняет операцию

```
[src] - [dst]
```

и устанавливает флаги условий в соответствии с полученным результатом. Она не изменяет значения ни одного из операндов.

Еще одна полезная структура данных, очень похожая на стек, называется *очередью*. Данные помещаются в очередь и извлекаются из нее в порядке «первым вошел — первым вышел» (First In First Out, FIFO). Как правило, очередь растет в направлении увеличения адресов. В этом случае новые данные добавляются в ее конец, имеющий наибольший адрес, а извлекаются таковые из начала очереди, имеющего наименьший адрес.

SAFEPOP	Compare Branch>0	#2000,SP EMPTYERROR	Проверка того, не содержит ли указатель стека значение больше 2000. Если это так, значит, стек пуст, и осуществляется переход к подпрограмме EMPTYERROR, выполняющей соответствующее действие
	Move	(SP)+,ITEM	В противном случае элемент, расположенный на вершине стека, выталкивается, а затем помещается в память по адресу ITEM

а

SAFEPUSH	Compare Branch≤0	#1500,SP FULLERROR	Проверка того, не содержит ли указатель стека значение меньше или равное 1500. Если содержит, значит, стек полон. В таком случае выполняется переход к подпрограмме FULLERROR, осуществляющей соответствующее действие
	Move	NEWITEM,–(SP)	В противном случае в стек помещается элемент, хранящийся в памяти по адресу NEWITEM

б

Рис. 2.23. Контроль пустого и полного стеков при выполнении операций проталкивания и выталкивания элемента: выталкивание элемента из стека (*а*); помещение элемента в стек (*б*)

Между способами реализации стека и очереди имеются два важных различия. Один конец стека (его дно) зафиксирован, а другой перемещается то вверх то вниз по мере добавления и удаления данных. Для ссылки на вершину стека в каждый конкретный момент нужен один-единственный указатель. Иное дело — очередь. По мере добавления и удаления данных оба ее конца смещаются в направлении увеличения адресов. Поэтому приходится отслеживать местоположение обоих концов очереди, используя для этого два указателя.

Второе различие между стеком и очередью состоит в том, что без дополнительного управления очередь будет постоянно смещаться в памяти компьютера в направлении увеличения адресов. Один из способов удержания ее в фиксированной области заключается в использовании так называемого *циклического буфера*. Предположим, очереди выделена область памяти от адреса BEGINNING до адреса END. Первый элемент помещается в очередь по адресу BEGINNING, а последующим поочередно добавляющимся элементам присваиваются возрастающие адреса. Когда

конец очереди достигнет адреса END, в начале выделенной для нее области памяти может уже образоваться свободное пространство, поскольку время от времени какие-то элементы удаляются из очереди. В таком случае указателю конца очереди снова будет присвоен адрес BEGINNING, и процесс повторится. Как и в случае стека, при выполнении операций с очередью необходимо отслеживать моменты заполнения выделенной для нее области памяти и полной очистки очереди (см. упражнения 2.18 и 2.19).

2.9. Подпрограммы

Очень часто программа должна по многу раз выполнять определенную подзадачу, но с разными значениями данных. Такая подзадача обычно называется *подпрограммой*. Подпрограмма может, скажем, вычислять функцию sin или сортировать список в порядке возрастания или убывания значений.

Составляющий такую подпрограмму блок команд можно включать во все те места программы, где он должен выполняться. Однако на практике так никогда не поступают. Для экономии места в память помещают только одну копию блока команд, и любая программа, которой потребуется выполнить эту подпрограмму, просто переходит к ее начальному адресу. Такой переход называется *вызовом* подпрограммы и выполняется в помощью команды Call.

После реализации подпрограммы работа вызывающей ее программы должна быть продолжена. В таком случае говорят, что выполняется *возврат* из подпрограммы в вызывающую программу. Делается это с помощью команды Return. Поскольку подпрограмма может вызываться из нескольких разных мест основной программы, при ее вызове где-то должен сохраняться адрес возврата. Иными словами, для обеспечения правильного возврата из подпрограммы команда Call должна сохранить содержимое регистра PC.

Применяемый компьютером способ выполнения вызовов подпрограмм и возврата из таковых называется методом *связывания подпрограмм*. Простейший метод связывания подпрограмм заключается в сохранении адреса возврата в заданном месте, которым может быть специально выделенный для этого регистр. Такой регистр называется *регистром связи*. Когда работа подпрограммы завершается, команда Return возвращает управление вызывающей программе, выполняя неявный переход через регистр связи.

Особой разновидностью команды перехода является команда Call, выполняющая такие операции, как сохранение содержимого регистра PC в регистре связи и переход по указанному в команде целевому адресу. Команда Return также является разновидностью команды перехода, но она выполняет переход по адресу, заданному в регистре связи. Этот процесс проиллюстрирован на рис. 2.24.

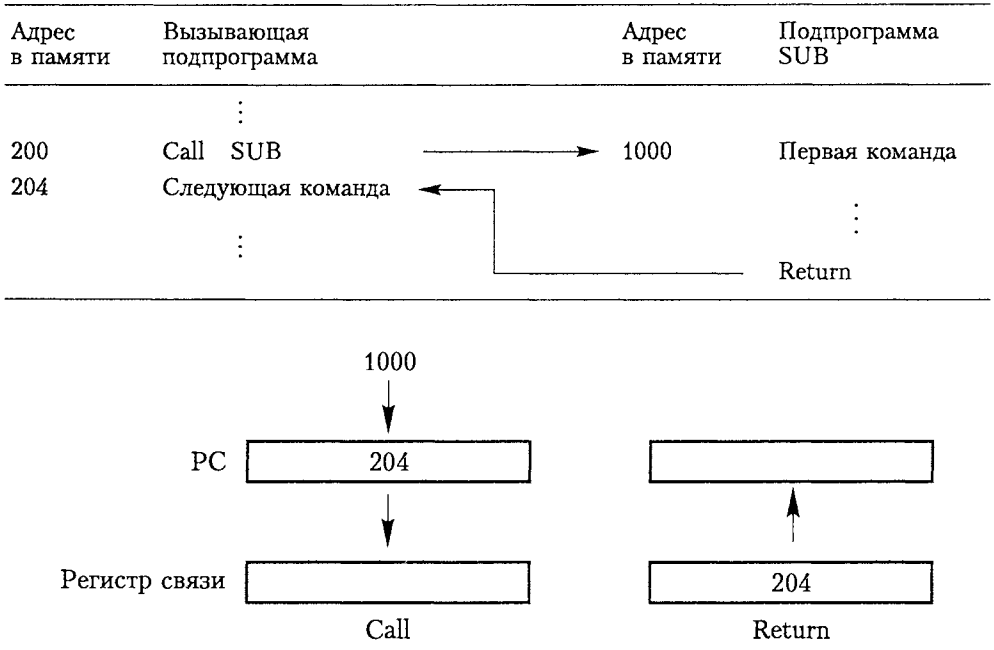


Рис. 2.24. Связывание подпрограммы через регистр связи

2.9.1. Вложенность подпрограмм и стек процессора

В программировании очень распространена практика вызова одних подпрограмм из других. Такие подпрограммы, вызываемые друг из друга, называются *вложенными*. Если вложенные вызовы будут реализованы по описанной выше технологии, вторая команда Call сохранит адрес возврата в регистре связи, уничтожив его текущее содержимое, после чего возврат в исходную программу станет не возможным. Поэтому перед вызовом очередной подпрограммы нужно где-то сохранять содержимое регистра связи.

Теоретически подпрограммы могут вкладываться на любую глубину. Рано или поздно последняя вызванная подпрограмма завершит свою работу и вернет управление вызвавшей ее подпрограмме. Необходимый для этого адрес возврата — это последний адрес, сохраненный в данной цепочке вызовов подпрограмм. Иными словами, адреса возврата сохраняются и используются в порядке LIFO (Last In First Out, что в переводе с английского значит «последним вошел — первым вышел»). Очевидно, что адреса возврата, связанные с вызовами подпрограмм, должны помещаться в стек. Многие процессоры делают это автоматически, как часть работы команды Call. Для использования в качестве указателя стека (SP) вызовов подпрограмм выделяется отдельный регистр, именуемый *стеком процессора*. Команда Call помещает в стек процессора содержимое регистра PC, а команда Return выталкивает из этого стека адрес возврата и помещает его в PC.

2.9.2. Передача параметров

Вызывая подпрограмму, программа должна передать ей параметры (операнды), которые будут использоваться в вычислениях, или же их адреса. Закончив свою работу, подпрограмма вернет другие параметры — результаты вычислений. Такой обмен информацией между вызывающей программой и подпрограммой называется *передачей параметров*. Передача параметров может выполняться несколькими способами. Например, параметры можно помещать в регистры или в память, откуда подпрограмма сможет их считать. В качестве альтернативы параметры можно поместить в стек процессора, используемый для хранения адресов возврата.

Использование регистров процессора — способ простой и эффективный. На рис. 2.25 показано, как реализовать программу, представленную на рис. 2.16 (выполняет сложение последовательности чисел), в виде подпрограммы с передачей параметров через регистры. Длина последовательности n, информация о которой хранится в памяти по адресу N, и адрес первого числа, NUM1, передаются подпрограмме через регистры R1 и R2. Вычисленная подпрограммой сумма возвращается вызывающей программе через регистр R0. Соответствующую часть вызывающей программы составляют первые четыре команды из числа представленных на рис. 2.25. Первые две команды загружают в регистры R1 и R2 значения N и NUM1. Команда Call выполняет переход к подпрограмме, начинающейся по адресу LISTADD. Кроме того, эта команда помещает в стек процессора адрес возврата из подпрограммы. Подпрограмма вычисляет сумму и помещает ее в регистр R0. После возврата из подпрограммы вызывающая программа сохраняет эту сумму в памяти по адресу SUM.

Вызывающая программа

```
            Move        N,R1        R1 играет роль счетчика
            Move        #NUM1,R2    R2 указывает на последовательность
            Call        LISTADD     Вызов подпрограммы
            Move        R0,SUM      Сохранение результата
            .
            .
            .
```

Подпрограмма

```
LISTADD     Clear       R0          Инициализация суммы значением 0
LOOP        Add         (R2)+,R0    Добавление числа из последовательности
            Decrement   R1
            Branch>0    LOOP
            Return                  Возврат в вызывающую программу
```

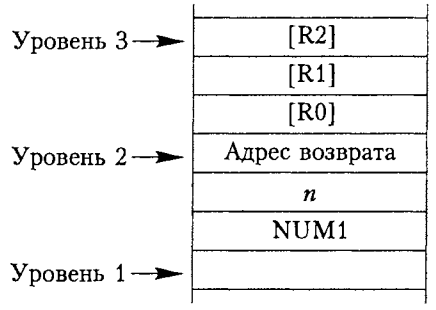
Рис. 2.25. Программа, представленная на рис. 2.16, реализованная как подпрограмма; параметры передаются через регистры

Если у подпрограммы много параметров, для их передачи может просто не хватить регистров общего назначения. С другой стороны, стек — структура гибкая, в него можно поместить много параметров. Следующий пример показывает, как выполняется передача параметров через стек. На рис. 2.26, *а* приведена та же программа, что и на рис. 2.16, но реализованная в виде подпрограммы LISTADD. Любая другая программа может вызвать указанную подпрограмму для сложения

последовательности чисел. Подпрограмме LISTADD передается адрес первого числа в последовательности и количество ее элементов. Подпрограмма выполняет сложение и возвращает полученную сумму. Ее параметры помещаются в стек процессора, на который указывает регистр SP. Предположим, что до вызова подпрограммы вершина стека располагается на уровне 1 (рис. 2.26, *б*). Вызывающая программа помещает в стек адрес NUM1 и значение *n* и вызывает подпрограмму LISTADD. Кроме того, команда Call помещает в стек адрес возврата из подпрограммы. Теперь вершина стека располагается на уровне 2.

```
(Предполагается, что вершина стека расположена на уровне 1)
            Move        #NUM1,-(SP)   Помещение параметров в стек
            Move        N,-(SP)
            Call        LISTADD       Вызов подпрограммы
                                      (вершина стека на уровне 2)
            Move        4(SP),SUM     Сохранение результата
            Add         #8,SP         Восстановление вершины стека
                                      (вершина стека на уровне 1)
            .
            .
            .
LISTADD     MoveMultiple R0-R2,-(SP)  Сохранение регистров
                                      (вершина стека на уровне 3)
            Move        16(SP),R1     Инициализация счетчика значением n
            Move        20(SP),R2     Инициализация указателя на
                                      последовательности чисел
            Clear       R0            Инициализация суммы значением 0
LOOP        Add         (R2)+,R0      Добавление числа из
                                      последовательности чисел
            Decrement   R1
            Branch>0    LOOP
            Move        R0,20(SP)     Помещение результата в стек
            MoveMultiple (SP)+,R0-R2  Восстановление регистров
            Return                    Возврат в вызывающую программу
```

а



б

Рис. 2.26. Программа, представленная на рис. 2.16, реализована как подпрограмма, с передачей параметров через стек: вызывающая программа и подпрограмма (*а*); вершина стека в разные моменты времени (*б*)

Подпрограмма использует три регистра. Поскольку в них могут содержаться данные, принадлежащие вызывающей программе, их содержимое сохраняется в стеке. Для помещения в стек содержимого регистров от R0 до R2 мы используем команду MoveMultiple. Подобные команды имеются у многих процессоров. Теперь вершина стека располагается на уровне 3. С помощью индексной адресации подпрограмма считывает из стека параметры *n* и NUM1. Значение указателя стека при этом не меняется, поскольку на вершине стека по-прежнему располагаются нужные нам элементы данных. Значение *n* загружается в регистр R1, который будет играть роль счетчика, а адрес NUM1 — в регистр R2, который будет служить указателем при сканировании списка значений. Регистр R0 должен содержать результат суммирования. Перед возвратом из подпрограммы содержимое регистра R0 помещается в стек, заменяя параметр NUM1, который нам больше не нужен. Затем из стека восстанавливается содержимое трех регистров, использовавшихся подпрограммой. Теперь верхним элементом стека является адрес возврата, расположенный на уровне 2. После возврата из подпрограммы вызывающая программа сохраняет результат в памяти по адресу SUM и возвращает вершину стека на ее исходный уровень, уменьшая значение SP на 8.

Передача параметров по значению и по ссылке

Обратите внимание на то, как передаются параметры NUM1 и *n* (рис. 2.25 и 2.26). Задачей подпрограммы является сложение последовательности чисел. Вместо того чтобы передавать реальные элементы последовательности, вызывающая программа передает подпрограмме адрес ее первого элемента. Эта технология называется *передачей по ссылке*. Второй параметр *передается по значению*, то есть подпрограмме передается реальное количество элементов, *n*, а не адрес того места, где эта информация хранится.

2.9.3. Стековый фрейм

На примере программы, представленной на рис. 2.26, давайте разберемся, как используется пространство стека. Во время выполнения подпрограммы применяемую ею информацию содержат шесть верхних элементов стека. Эта область составляет собственное рабочее пространство подпрограммы, создаваемое при ее вызове и освобождаемое, когда управление возвращается вызывающей программе. Она называется *стековым фреймом*. Если для локальных переменных подпрограммы требуется дополнительное пространство, его также можно выделить в стеке.

Пример типичного расположения информации в стековом фрейме приведен на рис. 2.27. В дополнение к указателю стека SP можно использовать еще один регистр, называемый *указателем фрейма* (Frame Pointer, FP). Он облегчает доступ к параметрам подпрограммы и ее локальным переменным. Эти локальные переменные используются только внутри подпрограммы, так что память для их хранения удобнее всего выделить прямо в стековом фрейме, связанном с данной подпрограммой. В нашем случае предполагается, что подпрограмма получает четыре параметра, применяет три локальные переменные и сохраняет содержимое регистров R0 и R1, которые она использует для своих нужд.

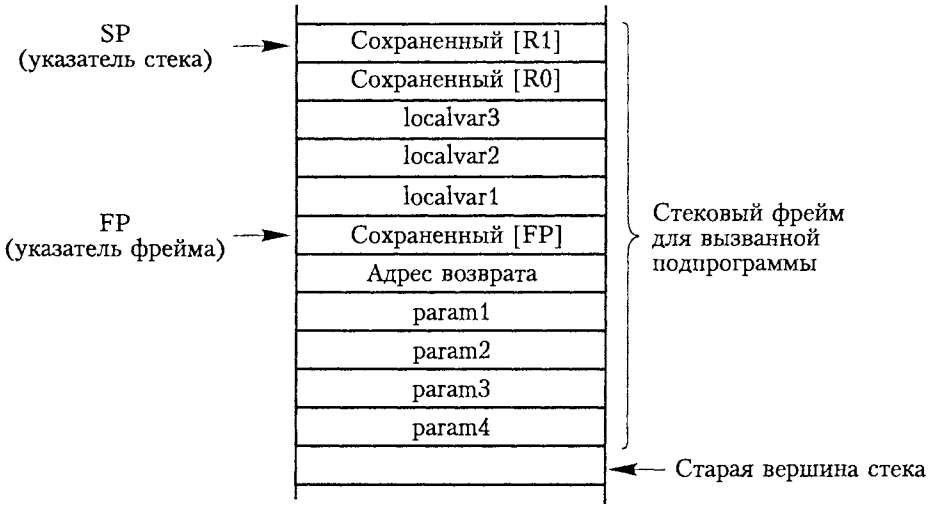


Рис. 2.27. Пример стекового фрейма

Поскольку регистр FP указывает на область памяти, расположенную непосредственно над сохраненным в стеке адресом возврата, с его помощью легко обращаться к параметрам и локальным переменным, применяя индексный режим адресации. Параметры адресуются так: 8(FP), 12(FP) и т. д., а локальные переменные так: −4(FP), −8(FP), ... Содержимое регистра FP в ходе выполнения подпрограммы остается неизменным, тогда как указатель стека SP должен всегда указывать на верхний элемент стека.

Теперь давайте посмотрим, что происходит с указателями SP и FP при создании, использовании и уничтожении стекового фрейма в результате вызова конкретной подпрограммы. Мы будем считать, что перед вызовом подпрограммы SP указывает на старую вершину стека.

Вызывающая программа помещает в стек четыре параметра. Затем выполняется команда Call, которая помещает в стек адрес возврата и передает управление подпрограмме. Теперь SP указывает на элемент стека, содержащий адрес возврата, и в следующий момент будет выполнена первая команда подпрограммы. Именно в это время указателю фрейма FP присваивается нужный адрес. Поскольку в качестве FP обычно выступает один их регистров общего назначения, он может содержать информацию, нужную вызывающей программе. Поэтому его содержимое тоже сохраняется в стеке. А поскольку на вершину стека, которая будет началом стекового фрейма, указывает SP, содержимое данного регистра копируется в FP.

Итак, первые две команды подпрограммы должны быть такими:

```
Move    FP,-SP
Move    SP,FP
```

После их выполнения SP и FP указывают на сохраненное содержимое регистра FP. Далее в стеке выделяется пространство для трех локальных переменных, для чего выполняется команда

```
Subtract #12,SP
```

Напоследок в стеке сохраняется содержимое регистров процессора R0 и R1. Теперь стековый фрейм сформирован и имеет такую структуру, как на рис. 2.27.

Далее подпрограмма, выполнив свою непосредственную задачу, выталкивает ранее сохраненные значения регистров R0 и R1 обратно в эти же регистры, удаляет из стекового фрейма локальные переменные с помощью команды

```
Add #12,SP
```

и выталкивает старое значение регистра FP обратно в этот регистр. Теперь SP указывает на адрес возврата и можно выполнить команду Return, возвращающую управление вызывающей программе.

За удаление параметров из стекового фрейма отвечает вызывающая программа. Некоторые из таких параметров могут содержать возвращенные подпрограммой результаты вычислений. После того как это будет сделано, указатель стека должен указывать на исходную вершину стека; мы же вернемся к тому, с чего начинали.

Стековые фреймы для вложенных подпрограмм

Стек — это структура данных, которая лучше всего подходит для хранения адресов возврата цепочки вложенных подпрограмм. Очевидно, что при вызове каждой из этих подпрограмм в стеке процессора формируется полный стековый фрейм. В этой связи важно отметить, что сохраненное содержимое регистра FP в текущем фрейме на вершине стека — это указатель стекового фрейма той подпрограммы, которая вызвала текущую подпрограмму.

В коде, приведенном на рис. 2.28, главная подпрограмма вызывает подпрограмму SUB1, которая, в свою очередь, вызывает подпрограмму SUB2. Стековые фреймы этих двух вложенных подпрограмм изображены на рис. 2.29. Все их параметры передаются через стек. На рисунках показаны только поток выполнения подпрограмм и их данные — реализацию основных задач мы опустили.

Порядок выполнения этой программы следующий. Главная программа помещает в стек два параметра, param2 и param1 (именно в таком порядке), а затем вызывает подпрограмму SUB1. Данная подпрограмма должна вычислить некоторое значение и передать его обратно главной программе через стек. В ходе вычислений SUB1 вызывает вторую подпрограмму, SUB2, выполняющую некоторую подзадачу. Подпрограмма SUB1 передает подпрограмме SUB2 один параметр, param3, и получает результат. После того как в SUB2 будет выполнена команда Return, подпрограмма SUB1 сохранит этот результат в регистре R2. Затем SUB1 продолжит свои вычисления, а закончив их, передаст ответ главной программе через стек. Когда управление будет возвращено главной программе, она сохранит полученный результат в памяти по адресу RESULT и продолжит свою работу со следующей команды. Этот процесс достаточно подробно описан в комментариях на рис. 2.28. Первым делом каждая подпрограмма устанавливает указатель фрейма (предварительно записав в стек его исходное содержимое) и сохраняет значения

тех регистров, которыми она собирается оперировать. Например, SUB1 использует четыре регистра, от R0 до R3, а SUB2 — два регистра, R0 и R1. Все указанные регистры, а также указатель стека восстанавливаются перед выходом из подпрограммы.

Адрес в памяти		Команды		Комментарии
Главная программа (Main)				
		⋮		
2000		Move	PARAM2,–(SP)	Помещение параметров в стек
2004		Move	PARAM1,–(SP)	
2008		Call	SUB1	
2012		Move	(SP),RESULT	Сохранение результата
2016		Add	#8,SP	Восстановление указателя стека
2020		Следующая команда		
		⋮		
Первая подпрограмма (SUB1)				
2100	SUB1	Move	FP,–(SP)	Сохранение указателя фрейма
2104		Move	SP,FP	Загрузка указателя фрейма
2108		MoveMultiple	R0–R3,–(SP)	Сохранение регистров
2112		Move	8(FP),R0	Считывание первого параметра
		Move	12(FP),R1	Считывание второго параметра
		⋮		
		Move	PARAM3,–(SP)	Помещение параметра в стек
2160		Call	SUB2	
2164		Move	(SP)+,R2	Выталкивание результата работы подпрограммы SUB2 в R2
		⋮		
		Move	R3,8(FP)	Помещение ответа в стек
		MoveMultiple	(SP)+,R0–R3	Восстановление регистров
		Move	(SP)+,FP	Восстановление указателя фрейма
		Return		Возвращение управления главной программе
Вторая подпрограмма (SUB2)				
3000	SUB2	Move	FP,–(SP)	Сохранение указателя фрейма
		Move	SP,FP	Загрузка указателя фрейма
		MoveMultiple	R0–R1,–(SP)	Сохранение регистров R0 и R1
		Move	8(FP),R0	Считывание параметра
		⋮		
		Move	R1,8(FP)	Помещение в стек результата работы подпрограммы SUB2
		MoveMultiple	(SP)+,R0–R1	Восстановление регистров R0 и R1
		Move	(SP)+,FP	Восстановление указателя фрейма
		Return		Возвращение управления подпрограмме SUB1

Рис. 2.28. Вложенные подпрограммы

Для загрузки параметров из стека и помещения результатов обратно в таковой используется индексный режим адресации с использованием указателя фрейма FP. В этих операциях, как и в случае стека, показанного на рис. 2.27, задается байтовое смещение 8, 12 и т. д. Обратите внимание на тот факт, что за удаление параметров из стека всегда отвечает вызывающая программа. Эту операцию выполняет команда Add в главной программе и команда Move, расположенная по адресу 2164 в подпрограмме SUB1.

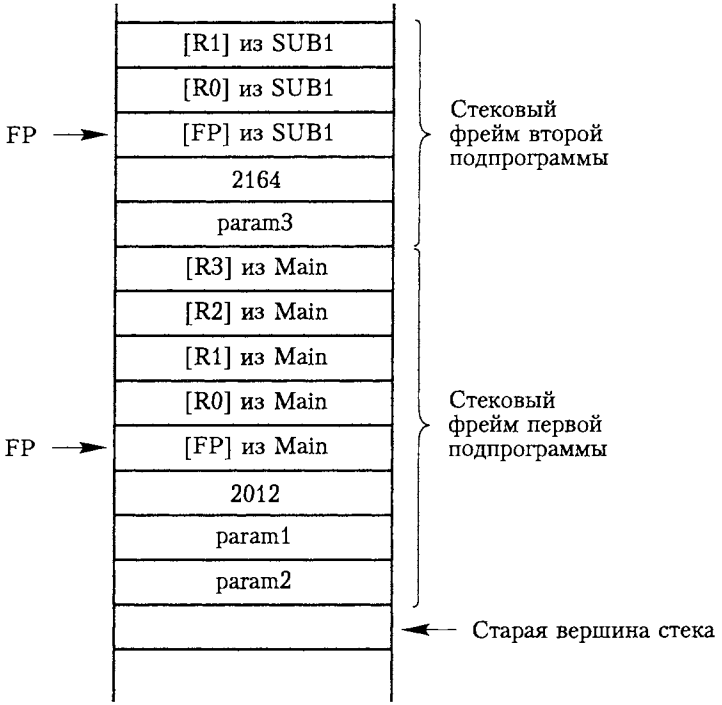


Рис. 2.29. Стековые фреймы программы, приведенной на рис. 2.28

2.10. Дополнительные команды

Итак, в этой главе вы познакомились с командами Move, Load, Store, Clear, Add, Subtract, Increment, Decrement, Branch, Testbit, Compare, Call и Return. Эти 13 команд в сочетании с приведенными в табл. 2.1 адресными режимами позволили нам написать ряд подпрограмм, демонстрирующих последовательность выполнения машинных команд, а также таких структур, как, например, ветвления. Кроме того, мы проиллюстрировали выполнение базовых операций ввода-вывода.

Но даже в таком маленьком наборе команд наблюдается избыточность. Так, команды Load и Store можно было бы заменить командой Move, а команды Increment и Decrement — соответственно командами Add и Subtract. Команду Clear также можно было бы заменить командой Move с непосредственно задаваемым операндом 0. Таким образом, нам вполне достаточно было бы восьми команд. Но

в системах реальных компьютеров избыточность встречается довольно часто. Многие из простых операций можно выполнить несколькими способами, правда, с разной эффективностью. В этом разделе вы познакомитесь еще с рядом важных команд, поддерживаемых большинством компьютеров.

2.10.1. Логические команды

Базовые строительные блоки логических схем (см. приложение А) реализуют такие выполняемые над отдельными битами логические операции, как И, ИЛИ и НЕ. Наряду с этим полезно иметь возможность производить указанные операции и программным путем, для чего используются команды, способные выполнять их над всеми битами слова или байта по отдельности и параллельно. Так, команда

```
Not  dst
```

дополняет все биты операнда, заменяя нули единицами, а единицы нулями. В разделе 2.1.1 было показано, что в результате добавления 1 к дополнению положительного числа со знаком до единицы получается отрицательная версия дополнения этого же числа до двух. Например, число +3 (0011) путем прибавления 1 к его дополнению до единицы преобразуется (рис. 2.1) в −3 (1101). Если число 3 содержится в регистре R0, данное преобразование выполняют команды

```
Not   R0
Add   #1,R0
```

Во многих компьютерах эту же задачу выполняет одна команда

```
Negate  R0
```

Теперь рассмотрим, как применяется логическая команда And, выполняющая побитовую операцию И над исходным и результирующим операндами. Предположим, что в 32-разрядном регистре R0 содержатся четыре символа ASCII. Нам требуется определить, является ли первый слева символ буквой Z. Если да, будет выполнен условный переход по адресу YES. В приложении Д указано, что ASCII-код буквы Z равен 01011010 или 5A в шестнадцатеричном формате. Нужное нам действие выполняют следующие три команды:

```
And          #$FF000000,R0
Compare      #$5A000000,R0
Branch=0     YES
```

Команда And очищает все биты трех расположенных справа символов, оставляя крайний слева символ нетронутым. Это результат использования непосредственно заданного исходного операнда, состоящего из восьми единиц слева и 24 нулей справа. Команда Compare сравнивает оставшийся символ с левого края регистра R0 с двоичным представлением символа Z. Если они совпадают, команда Branch выполняет переход по адресу YES.

Следует отметить, что команда And часто используется при необходимости очистить все разряды операнда, за исключением заданных разрядов. В нашем примере нужно оставить нетронутыми восемь крайних слева битов регистра R0.

2.10.2. Команды сдвига

Еще одна типичная задача программирования требует сдвига всех битов операнда вправо или влево на заданное количество разрядов. Процесс выполнения сдвига зависит от того, содержит ли операнд знак или какую-нибудь другую двоично-кодированную информацию. В общем случае, используется логический сдвиг. Для чисел производится арифметический сдвиг, при котором сохраняется знак числа.

Логический сдвиг

Для поддержки операции логического сдвига необходимы две команды: одна для выполнения сдвига влево (LShiftL), а другая — вправо (LShiftR). Эти команды сдвигают операнд на заданное количество разрядов, определяемое операндом count. Общий синтаксис команды, выполняющей логический сдвиг влево, таков:

```
LShiftL  count,dst
```

Операнд count может быть задан непосредственно или содержаться в регистре процессора. Освобождающиеся в результате сдвига разряды устанавливаются в 0, а сдвигаемые за границу операнда разряды отмечаются с помощью флага переноса C, а затем удаляются. Устанавливать флаг C особенно удобно при выполнении арифметических операций с большими числами, занимающими больше одного слова. На рис. 2.30, *а* показан пример сдвига содержимого регистра R0 влево на два разряда. Команда логического сдвига вправо работает точно так же (рис. 2.30, *б*).

Пример упаковки цифр

Давайте рассмотрим пример, иллюстрирующий использование обеих операций сдвига и логических операций. Предположим, что в памяти по байтовым адресам LOC и LOC+1 расположены две десятичные цифры, записанные с использованием кодов ASCII. Нам нужно представить каждую из них в 4-разрядном двоично-десятичном формате и сохранить обеих в одном байте по адресу PACKED. Результат, который мы получим, называется *упакованным двоично-десятичным форматом*. Из табл. Д.1 и Д.2 приложения Д видно, что четыре крайних справа бита ASCII-кода десятичной цифры соответствуют ее двоично-десятичному коду. Поэтому для выполнения нашей задачи достаточно извлечь четыре младших разряда из байтов, расположенных по адресам LOC и LOC+1, и объединить их в один байт по адресу PACKED.

Эту задачу выполняет представленная на рис. 2.31 последовательность команд, в которой для ссылки на ASCII-символы в памяти используется регистр R0, а для формирования двоично-десятичных кодов — регистры R1 и R2. Мы предполагаем, что байт, пересланный командой MoveByte из памяти в 32-разрядный регистр процессора, помещается у его правого края. Команда And используется для очистки всех разрядов регистра R2, за исключением четырех крайних справа. Обратите внимание, что промежуточный исходный операнд команды And задан как $F. Команда интерпретирует его как 32-разрядное двоичное число, содержащее 28 нулей в старших разрядах.

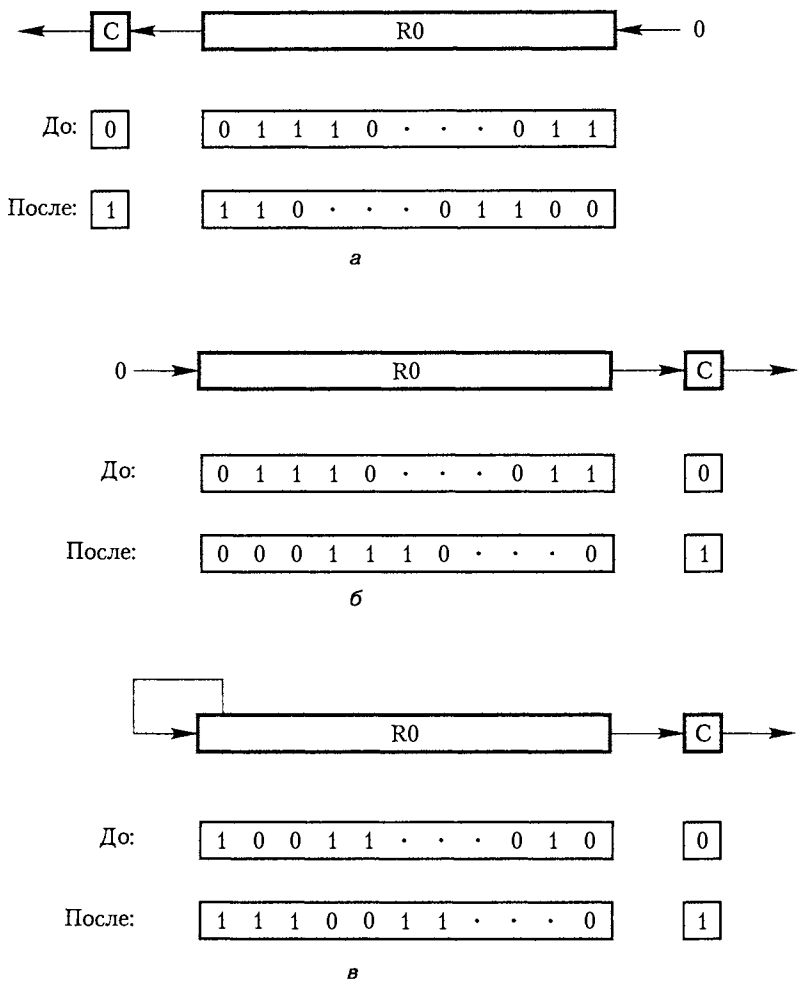


Рис. 2.30. Команды логического и арифметического сдвига:
логический сдвиг влево, LShiftL #2,R0 (*а*);
логический сдвиг вправо, LShiftR #2,R0 (*б*);
арифметический сдвиг вправо, AShiftR #2,R0 (*в*)

Move	#LOC,R0	R0 указывает на данные
MoveByte	(R0)+,R1	Загрузка первого байта в R1
LShiftL	#4,R1	Сдвиг первого байта влево на 4 разряда
MoveByte	(R0),R2	Загрузка второго байта в R2
And	#$F,R2	Очищение старших разрядов
Or	R1,R2	Объединение двоично-десятичных кодов цифр
MoveByte	R2,PACKED	Сохранение результата

Рис. 2.31. Подпрограмма, упаковывающая два двоично-десятичных числа

Арифметический сдвиг

Внимательно рассмотрев представление двоичного числа в форме дополнения до двух (рис. 2.1), вы поймете, что сдвиг числа на один разряд влево эквивалентен его умножению на 2, а сдвиг на один разряд вправо — его делению на 2. Конечно, при сдвиге влево может произойти переполнение, а при сдвиге вправо может потеряться конец числа. Еще одно важное наблюдение заключается в том, что при сдвиге вправо в освободившемся разряде должен быть повторен знаковый бит. Этим арифметический сдвиг вправо отличается от логического, в котором освобождающиеся разряды всегда заполняются нулями. Пример арифметического сдвига вправо (AShiftR) приведен на рис. 2.30, *в*. Арифметический сдвиг влево ничем не отличается от логического.

Циклический сдвиг

В операциях сдвига те биты, которые перемещаются за пределы операнда, оказываются просто потерянными. Сохраняется только последний сдвинутый бит, который копируется во флаг переноса C. Для сохранения всех битов операнда применяются операции циклического сдвига, перемещающие «выдвинувшиеся» с одного края разряды на другой край. Обычно компьютер поддерживает две версии циклического сдвига вправо и две версии циклического сдвига влево. В первой версии разряды операнда просто циклически сдвигаются, а во второй в сдвиге участвует еще и флаг C. Примеры всех четырех операций циклического сдвига приведены на рис. 2.32.

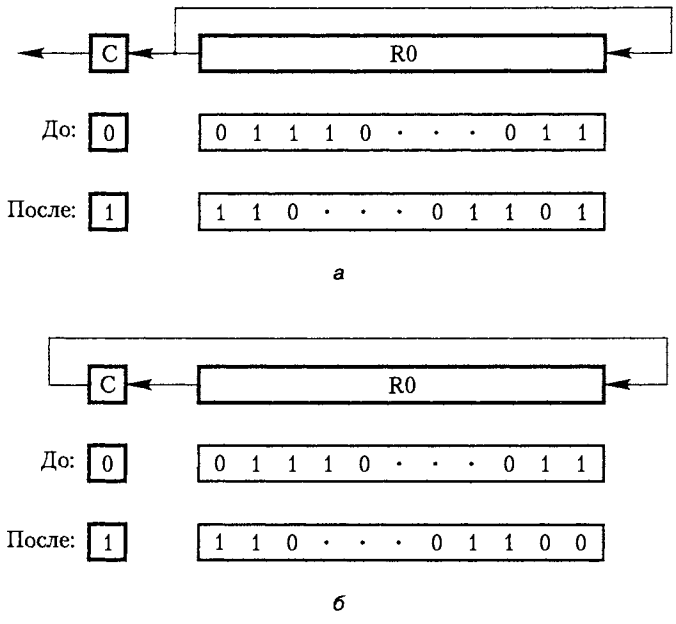


Рис. 2.32. Команды циклического сдвига: влево без переноса, RotateL #2,R0 (*а*); влево с переносом, RotateLC #2,R0 (*б*); вправо без переноса, RotateR #2,R0 (*в*); вправо с переносом, RotateRC #2,R0 (*г*)

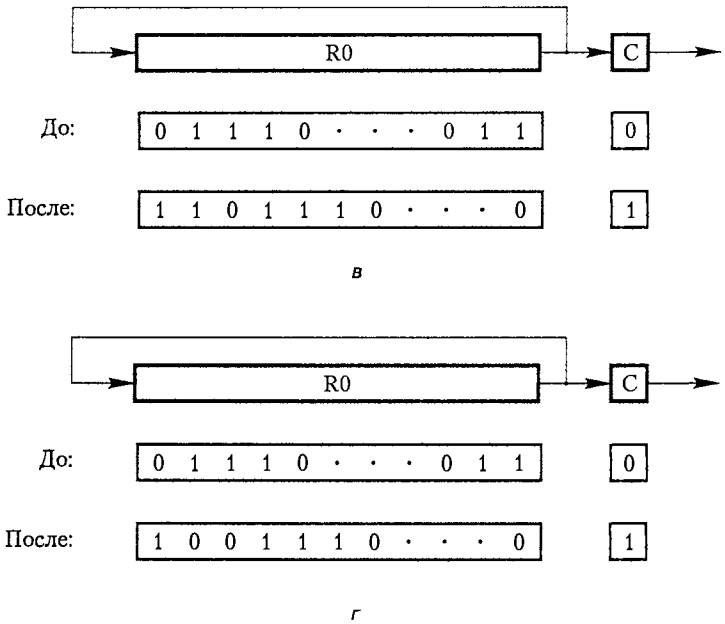


Рис. 2.32 (*продолжение*)

Обратите внимание, что в тех случаях, когда флаг C не участвует в операции, он, тем не менее, содержит последний бит, выдвинутый из операнда. Для определения операций циклического сдвига используются мнемонические обозначения RotateL, RotateLC, RotateR и RotateRC. Указанные команды в основном применяются в арифметических операциях, отличных от операций сложения и вычитания (см. главу 6).

2.10.3. Команды умножения и деления

Для умножения и деления двух целых чисел со знаком применяются машинные команды того же формата, что и команда Add. Так, команда

$$\text{Multiply } R_i,R_j$$

выполняет операцию

$$R_j \leftarrow [R_i] \times [R_j]$$

Произведение двух n-разрядных чисел может иметь размер до $2n$ разрядов. Таким образом, результат операции не всегда может поместиться в регистр Rj. Многие компьютеры поддерживают команду Multiply, которая вычисляет n младших разрядов произведения и помещает их в заданный регистр. Этого достаточно для тех ситуаций, когда заранее известно, что произведение имеет не более n разрядов. Для выполнения универсальной операции, формирующей полное произведение длиной в $2n$ разрядов, некоторые процессоры поддерживают другую команду, помещающую результат в два регистра, Rj и R(j+1).

В некоторых системах команд присутствует менее распространенная команда деления двух целых чисел со знаком

$$\text{Divide R}i\text{,R}j$$

выполняющая операцию

$$\text{R}j \leftarrow [\text{R}i] \,/\, [\text{R}j]$$

При этом частное помещается в регистр Rj, а остаток от деления либо помещается в регистр R(j+1), либо просто теряется.

Команды Multiply и Divide поддерживаются не всеми компьютерами. Те компьютеры, которые их не поддерживают, выполняют эти и другие арифметические операции с помощью последовательности элементарных команд, таких как Add, Subtract, Shift и Rotate. Вы поймете, как это делается, когда будете читать главу 6, посвященную способам реализации арифметических операций.

2.11. Примеры программ

В данном разделе вы познакомитесь с тремя примерами, иллюстрирующими принцип применения машинных команд. Они показывают, что собой представляют числовые (векторные) и нечисловые (выполняющие сортировку и операции со связным списком) приложения.

2.11.1. Программа вычисления скалярного произведения векторов

Первый пример представляет собой числовое приложение, подобное программе, приведенной на рис. 2.16, которая выполняла сложение чисел. В программах, осуществляющих обработку векторов и матриц, часто требуется вычислить скалярное произведение двух векторов. Предположим, у нас имеются векторы A и B длиной n. Их скалярное произведение определяется так:

$$\text{Скалярное произведение} = \sum_{i=0}^{n-1} \text{A}(i) \times \text{B}(i)$$

На рис. 2.33 приведена программа, вычисляющая скалярное произведение и сохраняющая результат в памяти по адресу DOTPROD. Первые элементы векторов, A(0) и B(0), хранятся по адресам AVEC и BVEC, а остальные — в следующих друг за другом словах.

Потребность в накоплении суммы произведений возникает во многих приложениях, выполняющих обработку сигналов. В таких приложениях один из векторов состоит из последних n отсчетов, поступивших на вход устройства обработки сигналов, а второй — из n весовых коэффициентов. Сумма их произведений составляет выходной сигнал устройства.

В системах некоторых компьютеров присутствует команда MultiplyAccumulate, объединяющая команды Multiply и Add, как в программе, представленной на рис. 2.33. В частности, такая команда имеется у процессора ARM (см. главу 3).

	Move	#AVEC,R1	R1 указывает на вектор A
	Move	#BVEC,R2	R2 указывает на вектор B
	Move	N,R3	R3 используется в качестве счетчика
	Clear	R0	В R0 накапливается сумма произведений
LOOP	Move Multiply	(R1)+,R4 (R2)+,R4	Вычисление произведения следующих компонентов
	Add	R4,R0	Прибавление к предыдущей сумме
	Decrement	R3	Уменьшение значения счетчика
	Branch>0	LOOP	Цикл
	Move	R0,DOTPROD	Сохранение скалярного произведения в памяти

Рис. 2.33. Программа для вычисления скалярного произведения двух векторов

2.11.2. Программа выполнения сортировки байтов

Рассмотрим программу для сортировки списка хранящихся в памяти байтов по алфавиту. Предположим, что список состоит из n байт, которые не обязательно должны быть разными, и что каждый байт содержит ASCII-код буквы латинского алфавита (из набора от A до Z). В кодировке ASCII, полностью приведенной в приложении Д, буквы A, B, ..., Z представлены 7-разрядными двоичными кодами. Если интерпретировать эти коды как числовые значения, будут получены последовательные целые числа. Когда символ ASCII записывается в память в виде байта, старший разряд этого байта устанавливается в 0. Таким образом, для сортировки списка букв по алфавиту достаточно отсортировать по возрастанию их числовые коды, рассматривая их как положительные числа.

Пусть наш список хранится в памяти по адресу от LIST до LIST + n − 1, где n — это 32-разрядное значение, находящееся по адресу N. Отсортированный список должен быть помещен на место исходного.

Для сортировки воспользуемся алгоритмом простого перебора. Сначала найдем наибольшее число и поместим его в конец списка по адресу LIST + n − 1. Затем в оставшемся списке из n − 1 элементов найдем наименьшее число и поместим его по адресу LIST + n − 2. Эта процедура будет повторяться до тех пор, пока мы не отсортируем весь список. На рис. 2.34, *а* приведена реализующая данный алгоритм программа на языке C, в которой список интерпретируется как одномерный массив с элементами LIST(0)...LIST(n−1). Для каждого подсписка LIST(j)...LIST(0) значение элемента LIST(j) сравнивается с каждым из остальных элементов подсписка. Если этот элемент оказывается больше, он меняется местами с числом LIST(j).

Программа на языке C обрабатывает список в направлении от конца к началу. Такой порядок облегчает завершение цикла в программе на машинном языке, поскольку цикл завершается тогда, когда текущий индекс становится равным нулю.

Программа на языке ассемблера, реализующая этот же алгоритм сортировки приведена на рис. 2.34, *б*. Комментарии к этой программе поясняют принцип использования различных регистров. В процессе сканирования подсписка текущее

максимальное значение хранится в регистре R3. Если значение элемента списка оказывается больше R3, оно меняется местами со значением в регистре R3 и записывается в LIST(j).

```
int main(int argc, char* argv[])
{
   for (j = n-1; j>0; j = j-1)
   {for k = j-1; k>=0; k = k-1)
      {if (LIST[k] > LIST[j])
         {TEMP = LIST[k];
          LIST[k] = LIST[j];
          LIST[j] = TEMP;
      }
   }
}
```

а

	Move	#LIST,R0	Загрузка списка в базовый регистр R0
	Move	N,R1	Инициализация индекса внешнего цикла
	Subtract	#1,R1	в регистре R1 значением $j = n - 1$
OUTER	Move	R1,R2	Инициализация индекса внешнего цикла
	Subtract	#1,R1	в регистре R2 значением $k = j - 1$
	MoveByte	(R0,R1),R3	Загрузка значения LIST(j) в регистр R3, содержащий текущее максимальное значение подсписка
INNER	CompareByte	R3,(R0,R2)	Если LIST(k) ≤ R3,
	Branch≤0	NEXT	не менять значения местами
	MoveByte	(R0,R2),R4	В противном случае поменять
	MoveByte	R3,(R0,R2)	местами LIST(j) и LIST(k) и загрузить
	MoveByte	R4,(R0,R1)	в R3 новое максимальное значение
	MoveByte	R4,R3	Регистр R4 выполняет функции переменой TEMP
NEXT	Decrement	R2	Уменьшение значения индексных
	Branch≥0	INNER	регистров R2 и R1, выполняющих
	Decrement	R1	функции счетчиков цикла, и, если цикл
	Branch>0	OUTER	не завершен, переход назад

б

Рис. 2.34. Программа, сортирующая байты методом простого перебора: на языке C (*а*); на языке ассемблера (*б*)

Управление ходом выполнения двух программ осуществляется по-разному. В управляющей инструкции if then в программе на языке C используется трехстрочное предложение then, меняющее местами элементы массива LIST(k) и LIST(j), если LIST(k) < LIST(j). В программе на языке ассемблера в случае, если

LIST(k) ≤ LIST(j), выполняется обход четырех команд, меняющих местами элементы списка. Благодаря этому программа получается более эффективной.

Если система команд компьютера позволяет перемещать данные из одного места памяти в другое, минуя регистры, тогда код обмена во внутреннем цикле, состоящий из четырех команд (рис. 2.34 *б*), можно сократить до трех команд:

```
MoveByte   (R0,R2),(R0,R1)
MoveByte   R3,(R0,R2)
MoveByte   (R0,R1),R3
```

Такую возможность поддерживает, в частности, процессор 68000 (см. главу 3).

Программа, приведенная на рис. 2.34, *б*, работает корректно лишь в том случае, если в списке содержатся хотя бы два элемента, поскольку проверка условия завершения цикла выполняется в конце цикла. Таким образом осуществляется как минимум один проход по циклу, независимо от значения n.

2.11.3. Связные списки

Во многих прикладных программах, не предназначенных для выполнения инженерных расчетов, упорядоченный список элементов данных должен быть представлен в памяти таким образом, чтобы в него легко было добавлять новые элементы или удалять из любого места ненужные, сохраняя определенный порядок. Такая структура более универсальна, чем стек или очередь (см. раздел 2.8), которые позволяют удалять и добавлять данные только в начало и конец списка. Рассмотрим пример. Список оценок, полученных студентами в результате трехкратного тестирования, на примере которого в разделе 2.5 рассматривался индексный режим адресации, содержит в первом слове каждой записи код студента (рис. 2.14). Предположим, мы хотим поддерживать этот список (он хранится в непрерывном блоке памяти) отсортированным по кодам студентов, а записи о студентах при этом могут добавляться и удаляться. Это позволит выводить на экран и печатать список в упорядоченном виде. Если после создания списка студент по какой-то причине будет исключен из группы, в списке останется пустая запись. В случае добавления в список новых оценок или печати такового эту пустую запись придется каждый раз пропускать. В еще более сложной ситуации мы окажемся, если после создания списка в группу будет включен новый студент. Чтобы список остался упорядоченным, нам придется сдвинуть в нем все записи начиная с номера, следующего за номером добавляемой записи. Точно так же при выходе студента из группы можно сдвинуть все записи, следующие за его удаленной записью, чтобы в списке не оставалось пробела.

Обеих этих сложных операций позволяет избежать структура данных, называемая *связным списком*. Как и в обычном списке, каждая запись в связном занимает четыре слова, но последовательные записи необязательно должны занимать последовательные блоки памяти. Для того чтобы связать записи в упорядоченный список, в каждую из них включают поле *указателя* длиной в одно слово, содержащее адрес следующей записи списка. Схематическое представление такой структуры данных показано на рис. 2.35, *а*. Первую запись связного списка иногда называют его «головой», а последнюю — «хвостом».

Для того чтобы в список добавить новую запись, между записями i и $i + 1$, нужно скопировать поле указателя из записи i в новую запись, а адрес новой записи поместить в поле указателя записи i. Эта операция схематически показана на рис. 2.35, *б*. Для удаления записи i значение из ее поля указателя копируется в поле указателя записи $i - 1$.

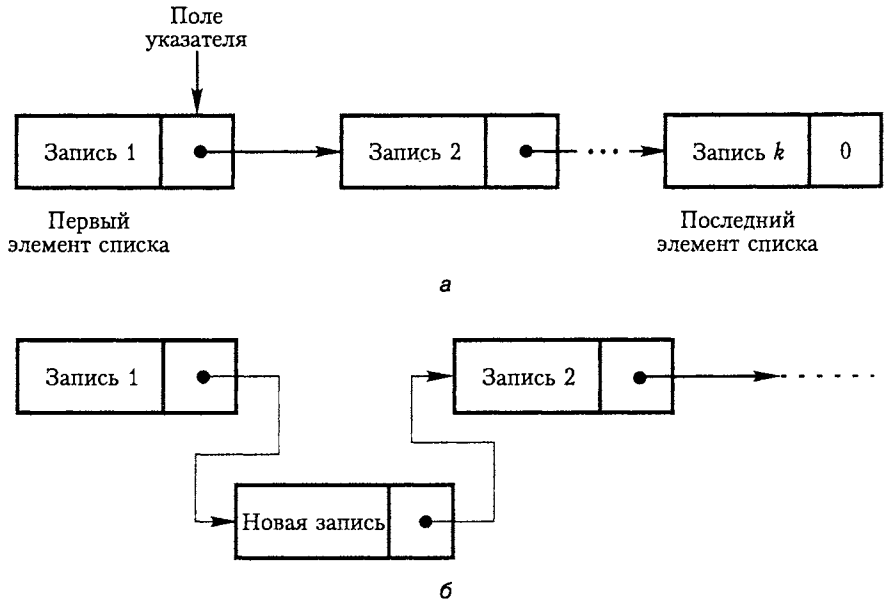


Рис. 2.35. Связный список: структура связного списка: (*а*); добавление новой записи между записями 1 и 2 (*б*)

На рис. 2.36 приведен пример записей с результатами тестов, связанных между собой по принципу связного списка и отсортированных по кодам студентов.

Длина каждой записи теперь составляет пять слов. В первом слове, определенном в качестве *ключевого* поля, содержится код студента, во втором — поле указателя, а остальные три слова хранят результаты тестов. Если слово имеет длину 32 разряда, для списка выделяется область памяти длиной 2000 байт начиная с адреса 1000. В таком случае каждая запись занимает 20 байт и выделенной области хватает для хранения 100 записей. Студенту, который включается в группу, назначается один из блоков памяти длиной 5 слов. Записям удобно назначать адреса 1000, 1020, 1040, ..., 2980, но так делать необязательно. Кроме того, никакой связи между кодами студентов и порядком их включения в группу не существует. Таким образом, блоки записей, упорядоченные по кодам студентов, будут совершенно непредсказуемым образом разбросаны по выделенной для списка области памяти и иметь произвольные адреса из диапазона от 1000 до 2980.

Запись с наименьшим кодом находится в начале списка, а с наибольшим — в конце. При работе со списком адрес его начала удобно хранить в регистре процессора, называемом *указателем начала списка*. В нашем примере это адрес 2320.

Адрес 1040 в поле указателя первой записи определяет местоположение второй записи. В поле указателя второй записи хранится адрес третьей записи — 1200. Поле указателя последней записи содержит значение 0, обозначающее конец списка. Если список пуст, 0 содержит поле указателя его начала.

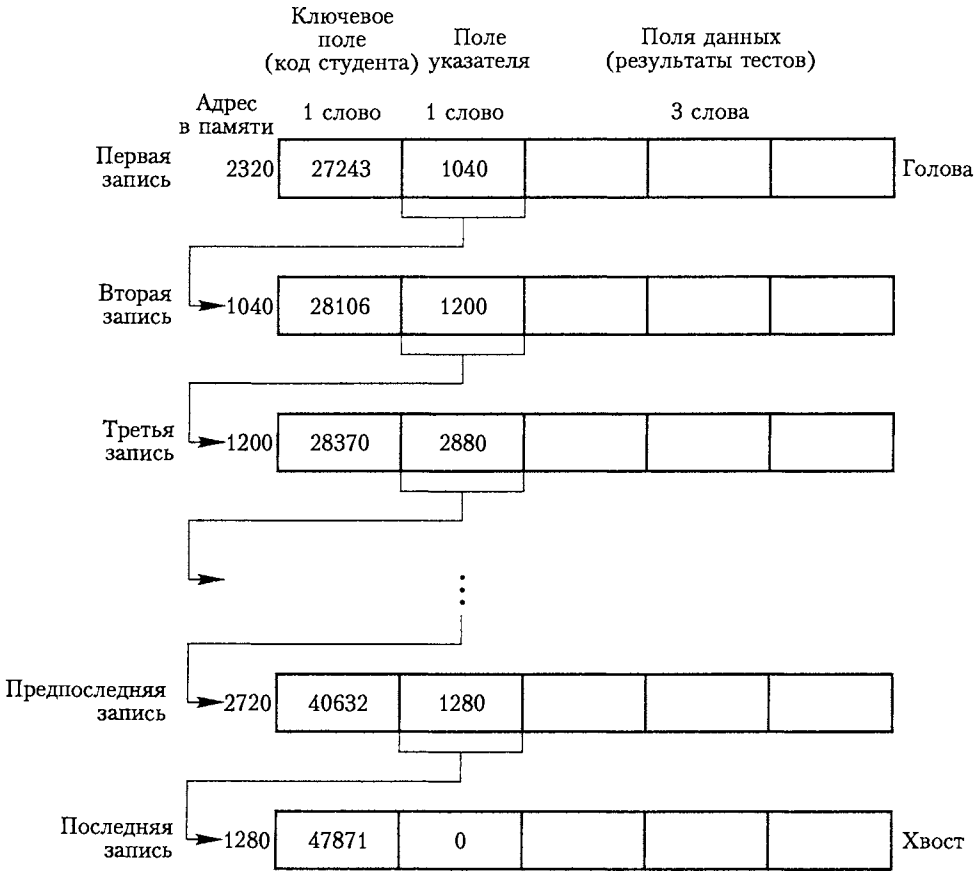


Рис. 2.36. Записи с результатами тестов, хранящиеся в виде связного списка

Вставка новой записи

А теперь мы более подробно рассмотрим процесс добавления новой записи в список, показанный на рис. 2.36. Предположим, что новая запись имеет код 28241, а следующий доступный блок памяти располагается по адресу 2960. Мы сканируем список записей начиная с головной, пока не достигнем записи, код которой будет больше кода добавляемой записи. Это запись с кодом 28370, расположенная по адресу 1200. Теперь введем в поле указателя новой записи значение 1200, а адрес новой записи, 2960, запишем в поле указателя предыдущей записи, расположенной по адресу 1040, заменив тем самым исходное значение 1200. Теперь новая

запись помещена на третье место списка, между второй и третьей записями исходного списка.

Выполняющая эту операцию подпрограмма приведена на рис. 2.37. Она состоит из трех частей, обрабатывающих три возможные ситуации, а именно: текущий список пуст, новая запись становится первой записью непустого списка и новая запись добавляется в любое другое место списка после первой записи. Последняя ситуация включает и тот случай, когда новая запись становится последней записью списка.

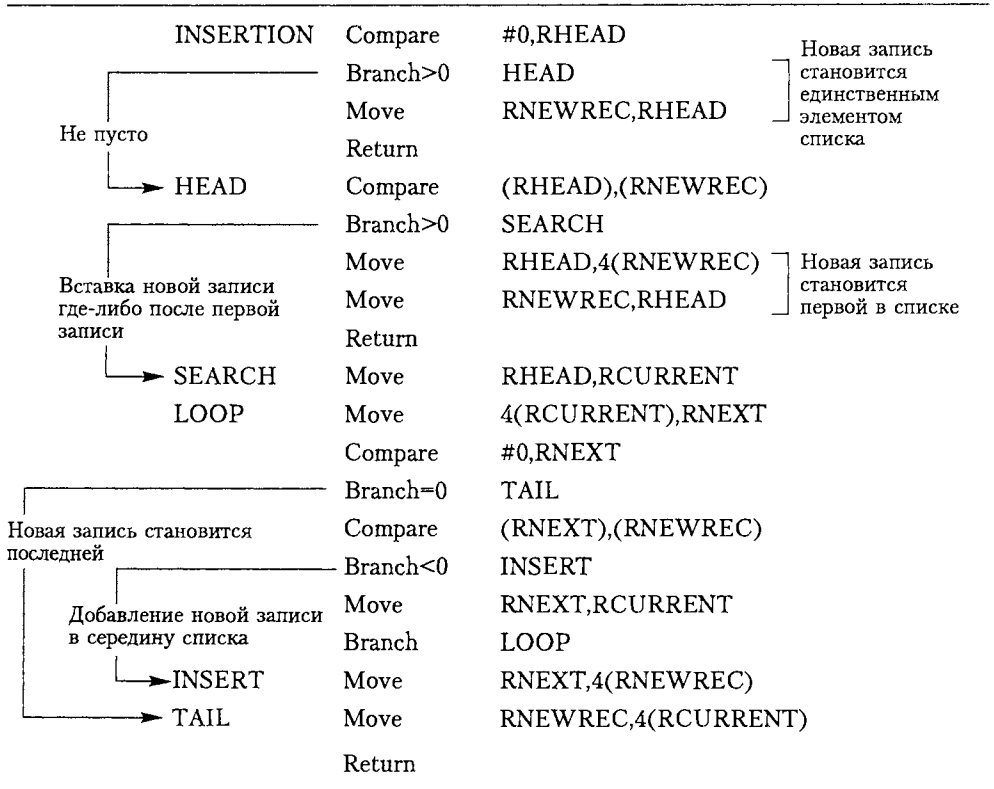


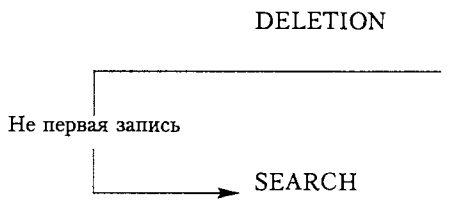
Рис. 2.37. Подпрограмма для добавления новой записи в связный список

Посмотрим, как подпрограмма обрабатывает три указанные ситуации. В ней используется целый ряд регистров процессора, которым мы, для того чтобы облегчить понимание программы, присвоили другие имена, более информативные по сравнению с традиционными R0, R1, R2 и т. д. Указатель начала списка мы назвали RHEAD, а регистр, содержащий адрес новой записи, — RNEWREC. Еще два регистра, RCURRENT и RNEXT, во время сканирования списка содержат адреса текущей и следующей записей. Поле указателя в новой записи первоначально установлено в 0. Если она становится последней записью списка, значение этого поля не меняется.

Первая пара команд сравнения и перехода проверяет, не пуст ли список. Если список пуст (RHEAD содержит 0), новая запись становится единственным элементом списка и ее адрес просто записывается в RHEAD, после чего выполняется команда возврата. В противном случае вторая пара команд сравнения и перехода проверяет, станет ли новая запись первой в списке. Если станет, две следующие команды пересылки изменяют содержимое ее поля указателя и регистра RHEAD, после чего выполняется команда возврата. Если новая запись не станет первой в списке, последняя часть подпрограммы определяет то место списка, куда она должна быть вставлена. Запись вставляется в нужное место с применением двух последних команд пересылки или добавляется в конец списка последней командой пересылки. Для того чтобы упростить эту подпрограмму, мы не стали приводить команды сохранения и восстановления регистров.

Удаление записи

Удаление существующей записи из связного списка — операция более простая. Нам нужно просканировать список и найти в нем запись с заданным кодом, а потом просто изменить значение указателя в предшествующей записи. Подпрограмма, выполняющая эту операцию, приведена на рис. 2.38.




```
            DELETION    Compare     (RHEAD),RIDNUM
                        Branch>0    SEARCH
                        Move        4(RHEAD),RHEAD
                        Return
            SEARCH      Move        RHEAD,RCURRENT
            LOOP        Move        4(RCURRENT),RNEXT
                        Compare     (RNEXT),RIDNUM
                        Branch=0    DELETE
                        Move        RNEXT,RCURRENT
                        Branch      LOOP
            DELETE      Move        4(RNEXT),RTEMP
                        Move        RTEMP,4(RCURRENT)
                        Return
```

Рис. 2.38. Подпрограмма для удаления записи из связного списка

Мы предполагаем, что в регистре RIDNUM содержится код удаляемой записи, а регистры RHEAD, RCURRENT и RNEXT выполняют те же функции, что и при добавлении записи. Первая пара команд сравнения и перехода проверяет, является ли подлежащая удалению запись первой записью списка. Если да, эта запись удаляется, для чего значение ее поля указателя копируется в RHEAD. Причем, если первая запись была единственной в списке, ее поле указателя содержит 0. Поэтому копирование этого указателя в регистр RHEAD означает, что теперь

список пуст. Регистры RCURRENT и RNEXT применяются для сканирования списка в поисках удаляемой записи. Когда эта запись будет найдена, вторая команда сравнения и перехода передаст управление по адресу DELETE. Запись, на которую указывает регистр RNEXT, будет удалена из списка, для чего ее поле указателя будет скопировано в поле указателя предыдущей записи, на которую указывает RCURRENT. Последние две команды пересылки выполняют такое копирование через регистр RTEMP. Если компьютер поддерживает непосредственное копирование из памяти в память, эти две команды можно заменить одной:

```
Move 4(RNEXT),4(RCURRENT)
```

Обработка ошибок

Подпрограммы добавления и удаления записи, показанные на рис. 2.37 и 2.38, не учитывают, что возможны две такие ситуации, из-за которых в них наверняка произойдут ошибки. Подпрограмма добавления записи предполагает, что в списке нет записи с таким кодом, как у новой записи, а подпрограмма удаления предполагает, что в списке такая запись имеется. Модификации этих подпрограмм с учетом двух указанных ситуаций посвящены упражнения 2.23 и 2.24.

2.12. Кодирование машинных команд

Итак, вы познакомились со множеством полезных команд и адресных режимов. Эти команды определяют действия, которые должны быть произведены процессором для выполнения конкретных задач. В предыдущих разделах мы часто называли их машинными командами. На самом же деле та форма, в которой приводись эти команды, близка к языку ассемблера — мы лишь пытались избежать использования сокращенных обозначений команд, так как они, во-первых, трудны для запоминания, а во-вторых, специфичны для конкретных процессоров. Чтобы процессор мог выполнить команду, ее необходимо закодировать в компактном двоичном формате. Именно такие закодированные команды следует называть *машинными командами*. А команды с символическими именами и сокращениями называются *командами на языке ассемблера*; программа-ассемблер преобразует их перед исполнением в машинные инструкции, как рассказывалось в разделе 2.6.

В предыдущих разделах мы, чтобы облегчить восприятие излагаемого материала, предполагали, что все команды имеют длину в одно слово. Поскольку речь обычно шла о 32-разрядных словах, считалось, что этой длины достаточно для представления всей необходимой информации. Теперь пришло время проверить это предположение.

Вам уже известны команды, выполняющие такие операции, как сложение, вычитание, пересылка, сдвиг и переход. В этих командах могут использоваться операнды разного размера, в частности 32- и 8-разрядные числа, 8-разрядные ASCII-коды символов. Типы выполняемых операций и их операндов могут определяться в двоичной спецификации, называемой *кодом операции* данной команды. Предположим, для кода операции выделяется 8 разрядов, что позволяет закодировать 256 разных команд. После этого в машинной команде для записи остальной информации остается 24 разряда.

Рассмотрим несколько типичных случаев. В команде

```
Add  R1,R2
```

помимо кода операции должны быть заданы два регистра, R1 и R2. Если процессор имеет 16 регистров, для идентификации каждого их них нужно 4 разряда. Кроме того, необходимо указать, что для каждого из операндов используется регистровый режим адресации, для чего потребуется еще несколько разрядов.

В команде

```
Move  24(R0),R5
```

16 разрядов необходимы для задания кода операции и двух регистров, а еще несколько разрядов для указания того, что для исходного операнда используется индексный режим адресации, а значение индекса составляет 24. Предположим, что для идентификации одного их перечисленных в табл. 2.1 режимов адресации необходимо иметь 3 разряда. Тогда в команде с двумя операндами для определения способов их адресации будет задействовано 6 разрядов. Для записи значения индекса остается 10 разрядов. Если их хватит для представления адекватного диапазона чисел со знаком, значит, команда поместится в 32-разрядное слово.

В команде сдвига

```
LShiftR  #2,R0
```

и команде пересылки

```
Move  #$3A,R1
```

необходимо предусмотреть разряды для записи непосредственно задаваемых значений 2 и $3A — это помимо 18 разрядов, используемых для определения кода операции, режимов адресации и регистра. Поэтому размер заданного операнда ограничен 14 разрядами.

Рассмотрим следующую команду перехода:

```
Branch>0  LOOP
```

И здесь код операции занимает 8 разрядов, а для смещения перехода остается 24 разряда. Поскольку смещение задается как число в формате дополнения до двух, целевой адрес перехода не должен отстоять от команды перехода более чем на 2^{23} байта. Для того чтобы выполнить переход к команде вне этого диапазона, нужно применить другой режим адресации, например абсолютную или регистровую косвенную адресацию. Команды перехода, в которых используются эти режимы, часто называют jump-командами.

Во всех рассмотренных примерах команды помещались в 32-разрядное слово. Возможный формат этого слова приведен на рис. 2.39, *а*. В нем восемь разрядов отведено для кода операции и по семь разрядов на каждый из двух операндов. Семиразрядное поле идентифицирует режим адресации операнда и используемый регистр (если таковой имеется). В поле «Другая информация» можно задать такие дополнительные сведения, как значение индекса или значение операнда.

Но как быть, если нам потребуется расположенный в памяти операнд задать с помощью абсолютного режима адресации? В команде

Move R2,LOC

для кода операции, адресных режимов и регистра требуется 18 разрядов. Следовательно, для адреса LOC остается всего 14 разрядов, чего, конечно же, совершенно недостаточно. Если в команде нужно задать полный 32-разрядный адрес, единственным решением может стать увеличение длины команды еще на одно слово, которое и будет содержать адрес. Формат такой команды показан на рис. 2.39, *б*. Первое слово в ней точно такое же, как на рис. 2.39, *а*, но второе содержит полный адрес операнда в памяти компьютера. В этом формате можно задавать команды наподобие следующей:

And #$FF000000,R2

В данном случае второе слово будет содержать непосредственно заданный 32-разрядный операнд.

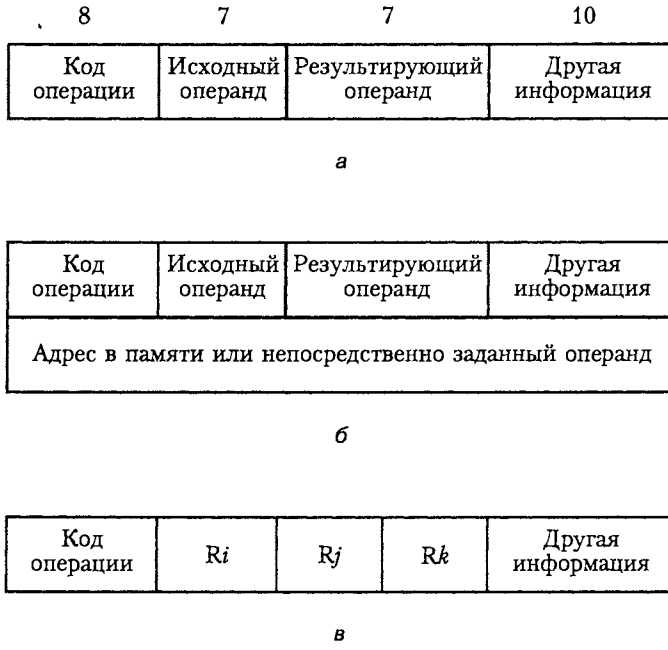


Рис. 2.39. Кодирование команд в 32-разрядных словах: команда длиной в одно слово (*а*); команда длиной в два слова (*б*); команда длиной в три слова (*в*)

Для хранения команды, в которой оба операнда заданы в абсолютном режиме, например такой:

Move LOC1,LOC2

потребуется не одно, а два дополнительных слова.

Как видите, мы пришли к выводу о необходимости использовать команды переменной длины, размер которых зависит от количества операндов и применяемых режимов адресации. Задействуя несколько слов, можно реализовать довольно сложные команды, напоминающие инструкции языков высокого уровня. Компьютеры, поддерживающие такие команды, называются *компьютерами с полным набором команд* (Complex Instruction Set Computer, CISC).

Существует и кардинально иная альтернатива этому подходу. Если решить, что все команды должны помещаться в одно 32-разрядное слово, в них просто невозможно будет использовать 32-разрядные адреса или 32-разрядные непосредственно задаваемые операнды. Но при этом система команд все равно может быть достаточно функциональной, если в ней будут интенсивно использоваться регистры процессора. Таким образом может поддерживаться команда

```
Add  R1,R2
```

но не команда

```
Add  LOC,R2
```

Вместо последней можно использовать команду

```
Add  (R3),R2
```

предварительно загрузив адрес LOC в регистр R3. В этом случае регистр R3 будет выполнять роль указателя на заданное место памяти.

Возникает вопрос: как загрузить 32-разрядный адрес в регистр, который должен служить указателем? Можно сделать так, чтобы ассемблер поместил нужный адрес в виде слова в область данных, расположенную рядом с программой. В таком случае для загрузки адреса можно использовать относительную адресацию. При этом нужно позаботиться о том, чтобы поле индекса в команде Load было достаточно большим для размещения в нем нужного адреса. Вторая возможность заключается в том, чтобы с помощью логических операций и операции сдвига сформировать в регистре 32-разрядный адрес, задавая его по частям, размеры которых достаточно малы для непосредственного задания в командах. Эта технология подробно рассматривается в той части главы 3, которая посвящена процессору ARM. Все команды данного процессора имеют 32-разрядный формат.

Компьютеры, размер команд которых ограничен одним словом, называются *компьютерами с сокращенным набором команд* (Reduced Instruction Set Computer, RISC). Системы команд типа RISC имеют и другие ограничения. В частности, все операции выполняются в них только над данными, хранящимися в регистрах процессора. Это означает, что для реализации приведенной выше операции сложения необходимы две команды:

```
Move  (R3),R1
Add   R1,R2
```

Если в команде Add задаются два регистра, она занимает даже меньше одного слова, и становится возможным использование команды с тремя операндами:

```
Add  R1,R2,R3
```

выполняющей операцию

$$R3 \leftarrow [R1] + [R2]$$

Возможный формат этой команды приведен на рис. 2.39, *в*. Конечно, процессор должен уметь работать с командами, имеющими по три операнда. В тех системах команд, в которых все логические и арифметические операции выполняются только над регистрами, обращение к памяти производится с помощью лишь двух команд, а именно считывания данных в регистры из памяти и записи данных из регистров в память.

В системах типа RISC команд обычно бывает меньше, причем сами по себе они проще, чем команды типа CISC. Относительные преимущества двух описанных подходов мы обсудим в главе 8, посвященной архитектуре процессоров.

2.13. Резюме

В настоящей главе проанализированы принципы представления и выполнения команд на двух уровнях, а именно языка ассемблера и машинном. Причем сделано это было с точки зрения программиста. Основное внимание уделялось базовым принципам адресации и последовательности выполнения команд. Существующие типы операций, реализованных в системах команд современных компьютеров, были рассмотрены на примерах небольших программ. Вы познакомились с несколькими режимами адресации, а также с понятиями указателей и индексного доступа к данным. Были описаны базовые операции ввода-вывода и параллельно обсуждался процесс пересылки символов между клавиатурой, процессором и дисплеем, а также представлена концепция подпрограмм и перечислены команды, необходимые для ее реализации. Обсуждая методы связывания подпрограмм, мы рассмотрели в качестве примера приложение, поддерживающее структуру данных типа стека. Затронуты были и такие темы, как способы управления другими структурами данных, в частности очередью, массивами и связными списками. Напоследок были проанализированы два разных подхода к формированию набора машинных команд, известных как RISC и CISC. О том, как выбор каждого из этих конструкторских решений отражается на производительности компьютера, мы поговорим в главе 8.

Упражнения

2.1. Представьте десятичные значения 5, −2, 14, −10, 26, −19, 51 и −43 как 7-разрядные числа со знаком в следующих двоичных форматах:

а) значение со знаком;

б) дополнение до единицы;

в) дополнение до двух.

(В преобразовании десятичного формата в двоичный вам поможет приложение Д.)

2.2. а) Преобразуйте следующие пары десятичных чисел в 5-разрядные числа со знаком в формате дополнения до двух и сложите их. В каждом случае укажите, произошло ли переполнение.

5 и 10; 7 и 13; −14 и 11; −5 и 7; −3 и −8; −10 и −13.

б) Повторите упражнение 2.2, *а* для операции вычитания (вычтите второе число каждой пары из первого). В каждом случае укажите, произошло ли переполнение.

2.3. Если по какому-то адресу памяти располагается двоичное значение, попробуйте определить, что оно собой представляет: число или машинную команду?

2.4. В некотором байте памяти располагается двоичное значение 00101100. Что получится, если интерпретировать его как двоичное число? А если оно будет интерпретировано в качестве ASCII-кода?

2.5. Имеется компьютер с побайтово адресуемой памятью, разделенной на 32-разрядные слова, в которых используется обратный порядок байтов. Программа считывает вводимые с клавиатуры ASCII-символы и сохраняет их в последовательно расположенных байтах начиная с адреса 1000. Каким будет содержимое двух слов памяти, расположенных по адресам 1000 и 1004, после того, как будет введено имя Jonson?

2.6. Повторите упражнение 2.5 для прямого порядка байтов.

2.7. Программа считывает ASCII-символы, представляющие цифры десятичного числа, по мере ввода с клавиатуры и сохраняет их в последовательных байтах. Проанализируйте приведенную в приложении Д кодировку ASCII и скажите, какая операция необходима для преобразования каждого числа в эквивалентное ему двоичное число.

2.8. Напишите для процессора с одним сумматором программу, вычисляющую выражение

$$A \times B + C \times D$$

Предполагается, что этот процессор поддерживает команды Load, Store, Multiply и Add.

2.9. Список оценок студентов, представленный на рис. 2.14, несколько изменен: теперь он содержит по j оценок для каждого студента. Предположим, что в группе n студентов. Напишите программу на языке ассемблера для вычисления суммы оценок за каждый тест по всем студентам и сохранения этой суммы по адресам SUM, SUM +4, SUM + 8 и т. д. Количество тестов, j, больше количества регистров процессора, так что программа, аналогичная приведенной на рис. 2.15 (для трех тестов), в данном случае использоваться не может. Примените два вложенных цикла, как предлагалось в разделе 2.5.3. Внутренний цикл должен накапливать сумму за один тест, а внешний проходить по всем тестам. Предполагается, что количество тестов хранится в памяти по адресу J, перед адресом N.

2.10. а) Перепишите приведенную на рис. 2.33 программу вычисления скалярного произведения для системы команд, в которой арифметические

и логические операции могут применяться только к операндам в регистрах процессора. Для пересылки операндов между регистрами и памятью используются две команды: Load и Store.

б) Вычислите значения констант k_1 и k_2 в выражении $k_1 + k_2 n$, определяющем количество обращений к памяти, необходимое для выполнения программы из упражнения 2.10, *а*, включая и выборку слов команд. Предполагается, что каждая команда занимает одно слово.

2.11. Повторите упражнение 2.10 для компьютера с двухадресными командами, который может выполнять операции типа

$$A \leftarrow [A] + [B]$$

где A и B — это либо адреса в памяти, либо регистры процессора. Какому компьютеру потребуется меньше обращений к памяти? (Глава 8, посвященная конвейерной обработке команд, предлагает иной взгляд на этот вопрос.)

2.12. «Наличие большого количества регистров процессора позволяет сократить количество обращений к памяти, необходимых для выполнения сложных задач.» Придумайте простую задачу, доказывающую правильность этого утверждения по отношению к процессору с четырьмя регистрами в сравнении с процессором, имеющим только два регистра.

2.13. В регистрах R1 и R2 содержатся десятичные значения 1200 и 4600. Каков исполнительный адрес хранящегося в памяти операнда для каждой из следующих команд:

```
Load      20(R1),R5
Move      #3000,R5
Store     R5,30(R1,R2)
Add       -(R2),R5
Subtract  (R1)+,R5
```

2.14. Предположим, что список, приведенный рис. 2.14, хранится в памяти в виде связного списка, как на рис. 2.36. Напишите программу на языке ассемблера, которая выполняет ту же задачу, что и программа, представленная на рис. 2.15. Первая запись списка хранится по адресу 1000.

2.15. Имеется массив чисел $A(i,j)$, в котором i — это индекс строки, изменяющийся в пределах от 0 до $n-1$, а j — индекс столбца, изменяющийся в пределах от 0 до $m-1$. Массив хранится в памяти строка за строкой, а элементы каждой строки занимают m последовательных слов. Предположим, что память адресуется побайтово и длина слова составляет 32 разряда. Напишите подпрограмму для поэлементного сложения столбцов x и y с помещением сумм в столбец y. Индексы x и y передаются подпрограмме в регистрах R1 и R2. Параметры n и m передаются в регистрах R3 и R4, а адрес элемента A(0,0) — в регистре R0. Возможно использование любого из режимов адресации, перечисленных в табл. 2.1. В памяти должно располагаться не более одного операнда команды.

2.16. Каждая из двух последовательностей команд

```
ORIGIN      1000
DATAWORD    300
```

и

```
Move  #300,1000
```

записывает значение 300 в память по адресу 1000, но на это у них уходит разное время. Объясните, почему так происходит.

2.17. Регистр R5 используется программой в качестве указателя на вершину стека. Напишите последовательность команд с применением индексного, автоинкрементного и автодекрементного режимов адресации для выполнения перечисленных ниже задач.

а) Вытолкнуть первые два элемента из стека, сложить их и поместить результат в стек.

б) Скопировать пятый элемент от вершины стека в регистр R3.

в) Удалить из стека первые 10 элементов.

2.18. Реализуйте в памяти очередь (FIFO) байтов, занимающую фиксированную область размером k байт. Вам нужны два указателя, IN и OUT. Указатель IN должен отследить адрес, по которому в очередь будет помещен следующий элемент, а указатель OUT указывает на следующий байт, подлежащий удалению из очереди.

а) Когда данные добавляются в очередь, они поочередно помещаются по возрастающим адресам, пока не будет достигнут предел отведенной для очереди памяти. Что произойдет дальше, когда в очередь нужно будет добавить следующий элемент?

б) Найдите подходящее определение для указателей IN и OUT, соответствующее тому, на какие элементы структуры данных они указывают. Проиллюстрируйте свой ответ простой диаграммой.

в) Докажите, что если состояние очереди определяется только двумя указателями, ситуацию полного заполнения и исчезновения очереди выявить невозможно.

г) Что бы вы добавили в код для решения задачи 2.18, *в*?

д) Предложите процедуру добавления элементов в очередь и удаления элементов из таковой с использованием указателей IN и OUT.

2.19. Проанализируйте структуру очереди, описанной в упражнении 2.18. Напишите подпрограммы APPEND и REMOVE для пересылки данных между регистром процессора и очередью. Не забывайте при каждой операции аккуратно проверять и обновлять состояние очереди и указателей.

2.20. Проанализируйте перечисленные далее возможности сохранения адреса возврата из подпрограммы:

а) в регистре процессора;

б) в памяти, связанной с текущим вызовом, при условии, что для каждого вызова подпрограммы используется другой адрес памяти;

в) в стеке.

Какое из этих решений годится для вложенных вызовов подпрограмм и рекурсии (когда подпрограмма вызывает сама себя)?

2.21 Команда вызова подпрограммы сохраняет адрес возврата в регистре процессора RL, называемом регистром связи. Что необходимо для обеспечения возможности вложенных вызовов подпрограмм? Позволяет ли ваша схема подпрограммам вызывать самих себя?

2.22. Предположим, вы хотите организовать вызовы подпрограмм следующим образом: когда подпрограмме Main нужно вызвать подпрограмму SUB1, она вызывает промежуточную подпрограмму CALLSUB и передает ей в качестве параметра адрес SUB1 в регистре R1. Подпрограмма CALLSUB сохраняет адрес возврата в стеке, предварительно убедившись, что верхняя граница стека еще не достигнута. Затем она передает управление подпрограмме SUB1. Для возврата в вызывающую программу подпрограмма SUB1 вызывает еще одну промежуточную подпрограмму, RETRN. Эта подпрограмма убеждается, что стек не пуст, и использует его верхний элемент для возврата в исходную вызывающую программу.

Напишите подпрограммы CALLSUB и RETRN, предполагая, что команда вызова подпрограммы сохраняет адрес возврата в регистре связи RL. Адреса верхней и нижней границ стека записываются в память по адресам UPPERLIMIT и LOWERLIMIT.

2.23. Подпрограмма, вставляющая элемент в связный список (рис. 2.37), не проверяет, присутствует ли в списке код добавляемой записи. Что происходит, если такая проверка не выполняется? Модифицируйте подпрограмму таким образом, чтобы в описанном случае она возвращала в регистре ERROR адрес записи с тем же кодом или 0, если добавление выполнено успешно.

2.24. В представленной на рис. 2.38 подпрограмме, выполняющей удаление записи из связного списка, предполагается, что в списке обязательно имеется запись с кодом, указанным в регистре RIDNIM. А что если такой записи не существует? Модифицируйте подпрограмму таким образом, чтобы она возвращала в регистре RIDNIM значение 0, если удаление выполнено успешно, и оставляла содержимое этого регистра неизменным в противном случае.

Глава 3

Системы команд процессоров ARM, Motorola и Intel

- Архитектура процессоров ARM
- Архитектура процессоров Motorola 68000
- Архитектура процессоров Intel IA-32 Pentium

В главе 2 вы познакомились с понятиями «система команд» и «режим адресации», узнали, как команды выполняются в компьютере, рассмотрели множество примеров команд и программ, написанных на унифицированном языке ассемблера. Теперь вам предстоит узнать, как все это реализовано в архитектуре систем команд процессоров ARM, Motorola 68000 и Intel IA-32. Система команд процессора ARM является примером архитектуры RISC, а системы команд 68000 и IA-32 — примерами архитектуры CISC. Данная глава условно разбита на три части, и в каждой из них приводятся примеры, уже известные вам по главе 2, но переписанные для конкретных процессоров. Правда, на этот раз мы анализируем их очень сжато, поэтому для понимания излагаемого материала важно как следует усвоить основные идеи и хорошо понять описания программ, представленные в предыдущей главе. Кроме того, дополнительная информация о трех системах команд имеется в наших приложениях.

Система команд процессоров ARM

Компания Advanced RISC Machines (ARM) Limited разработала семейство процессоров и предоставила ряду компаний лицензии на их производство и использование в компьютерах и встроенных системах. ARM считается относительно молодой компанией. Она была создана на базе компании Acorn Computers, занимавшейся разработкой процессоров в начале 1980-х годов. Микропроцессоры ARM применяются главным образом в недорогих и не самых мощных встроенных системах, в частности в мобильных телефонах, коммуникационных модемах, системах управления автомобилями и карманных компьютерах. Подробное описание архитектуры и реализации процессоров ARM, много другой интересной информации вы найдете на web-узле компании, расположенном по адресу http://www.arm.com.

Во всех процессорах ARM используется одна и та же базовая система машинных команд с небольшими модификациями. В данной книге описывается ее реализация для процессора ARM7. В более поздние версии процессоров ARM были добавлены команды и функции, не связанные с обсуждаемыми в этой главе темами. О некоторых из них речь пойдет в главе 11. На примере программ из главы 2, переписанных на языке ассемблера процессоров ARM, мы рассмотрим важнейшие аспекты архитектуры этих процессоров.

3.1. Регистры, доступ к памяти и пересылка данных

В архитектуре процессоров ARM память адресуется побайтово, с помощью 32-разрядных адресов, а регистры процессора имеют длину 32 разряда. Для пересылки данных между памятью и регистрами процессора используются операнды длиной в байт (8 бит) и слово (32 бита). Адреса слов должны быть выровнены, то есть кратны 4. Поддерживается как прямой, так и обратный порядок байтов (см. раздел 2.2.2). Выбор нужного порядка осуществляется при помощи внешней входной управляющей линии процессора. Когда байт загружается из памяти в регистр процессора или записывается из регистра процессора в память, он всегда располагается в младшем байте регистра.

Доступ к памяти осуществляется только с применением команд считывания и записи. Все арифметические и логические команды оперируют лишь данными в регистрах процессора. Эта схема наиболее характерна для архитектуры RISC. В главе 8 будет рассказано, как она помогает упростить конструкцию процессора и повысить его производительность.

3.1.1. Регистры

Регистры процессора, предназначенные для использования прикладными программами, представлены на рис. 3.1. Всего имеется шестнадцать 32-разрядных регистров с именами от R0 до R15, из числа которых пятнадцать являются регистрами общего назначения (от R0 до R14), а один — регистром счетчика команд PC (Program Counter). В регистрах общего назначения могут храниться как адреса памяти, так и данные операндов. В регистре текущего состояния программы CPSR (Current Program Status Register), обычно называемом просто регистром состояния, содержатся флаги кодов условий (N, Z, C, V), флаги запрещения прерываний и биты состояния процессора. Информация, представляемая флагами кодов условий, описана в разделе 2.4.6. Об использовании битов режима процессора и битов запрещения прерывания рассказывается в контексте операций ввода-вывода и прерываний в главе 4. А в данной главе мы предполагаем, что процессор работает в пользовательском режиме и выполняет прикладную программу.

Помимо указанных существует 15 дополнительных регистров общего назначения, называемых *запасными* (banked). Они дублируют некоторые из регистров общего назначения и используются, когда процессор переключается в режим

супервизора или передает управление программе обработки прерываний. В этих режимах доступны также сохраненные копии регистра состояния. О запасных регистрах и копиях регистра состояния рассказывается в главе 4.

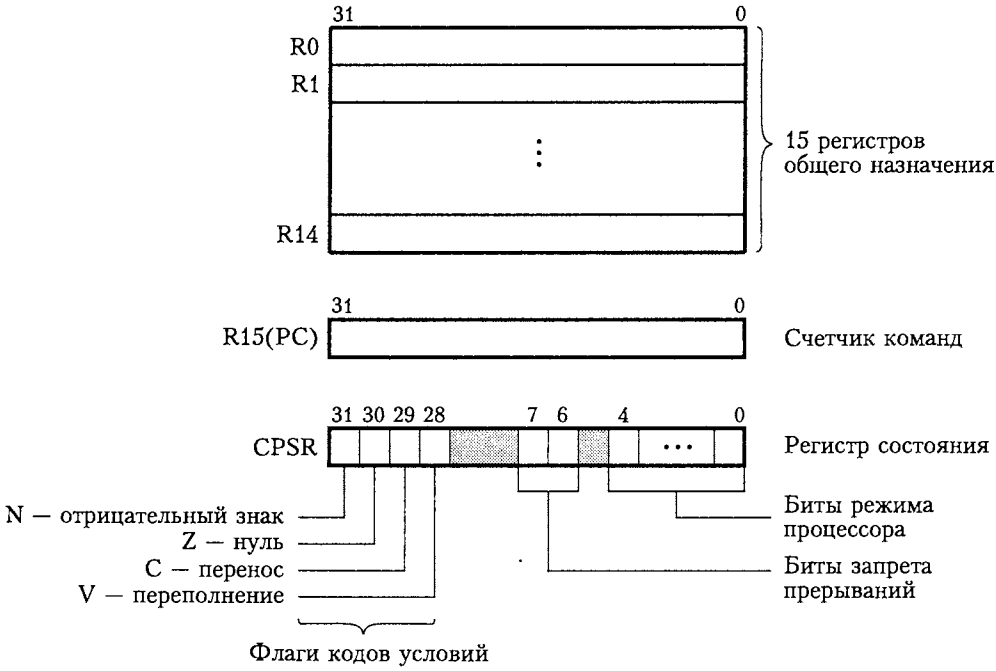


Рис. 3.1. Регистры процессора ARM

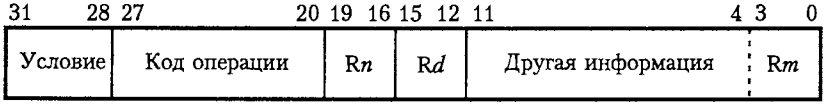


Рис. 3.2. Формат команды процессора ARM

3.1.2. Команды доступа к памяти и режимы адресации

Каждая команда в архитектуре ARM кодируется 32-разрядным словом. Кодировка команд унифицирована и типична для архитектуры RISC. Доступ к памяти осуществляется только командами Load и Store. Эти команды, команды пересылки, а также арифметические и логические команды имеют формат, показанный на рис. 3.2. Достаточно подробное его описание приведено в приложении Б. В команде задается код условного выполнения («Условие»), код операции, два или три регистра (R*n*, R*d*, R*m*) и некоторая другая информация. Если регистр R*m* не нужен, поле «Другая информация» расширяется до бита b_0. Посредством команды Load операнд пересылается из памяти в регистр общего назначения, указанный в 4-разрядном поле R*d*, а посредством команды Store — из регистра, имя которого

указано в поле R*d*, в память. Если операнд имеет длину 1 байт, он всегда находится в младшем байте регистра. Только в команде загрузки старшие 24 разряда регистра заполняются нулями.

Условное выполнение команд

Отличительной и несколько необычной особенностью процессоров ARM является то, что все его команды большей частью выполняются условно, с учетом заданного в команде условия. Команда действительно выполняется лишь в том случае, если флаги текущего состояния процессора удовлетворяют условию, заданному в разрядах b_{31-28}. В противном случае процессор переходит к следующей команде. Один из разрядов указывает, что команда будет выполняться всегда. С какой целью применяется условное выполнение всех команд, вы поймете из примеров, приведенных в разделе 3.7. А пока мы будем игнорировать эту функцию и считать, что команды содержат код «выполняется всегда».

Режимы адресации памяти

Основным методом адресации операндов в памяти является формирование исполнительного адреса операнда путем добавления к значению заданного в команде базового регистра R*n* значения смещения со знаком (рис. 3.2). Смещение либо непосредственно задается в 12 младших разрядах команды, либо содержится в третьем регистре, R*m*, указанном в четырех младших разрядах команды, b_{3-0}. Знак (направление) смещения содержится в поле кода операции.

Например, в команде загрузки

LDR R*d*,[R*n*,#смещение]

смещение (представленное числом со знаком) задано в режиме непосредственной адресации. Эта команда выполняет операцию

R*d* ← [[R*n*] + смещение]

Обратите внимание, что регистр назначения R*d* указывается первым. Этот порядок операндов противоположен используемому в главе 2. Команда

LDR R*d*,[R*n*,R*m*]

выполняет операцию

R*d* ← [[R*n*] + [R*m*]]

Поскольку содержимым регистра R*m* является величина смещения, то в том случае, если она отрицательна, перед именем данного регистра должен стоять знак «–». В главе 2 два указанных режима адресации определялись как индексная и базовая индексная. Нулевое смещение явно задавать не нужно. Поэтому команда

LDR R*d*,[R*n*]

выполняет операцию

R*d* ← [[R*n*]]

с использованием режима, определенного в главе 2 как косвенная адресация.

Мнемоническое обозначение кода операции LDR указывает, что из памяти в регистр должно быть загружено 32-разрядное слово. Байтовый операнд может быть загружен в младший байт регистра при помощи команды LDRB. При этом старшие разряды будут заполнены нулями.

Команды сохранения мнемонически обозначаются как STR и STRB. Так, команда

$$\text{STR } Rd,[Rn]$$

выполняет операцию

$$[Rn] \leftarrow [Rd]$$

пересылая в память операнд длиной в одно слово. Команда STRB пересылает в память один байт — младший байт регистра R*d*.

В документации ARM все три режима адресации и другие режимы, которые мы обсудим далее, называются индексными. Форма адресации, которая использовалась в нашем первом примере, называется преиндексной, поскольку исполнительный адрес операнда генерируется путем добавления величины смещения к содержимому базового регистра R*n*. При этом содержимое регистра R*n* не изменяется. Кроме того, поддерживаются режимы, подобные описанной в главе 2 автоинкрементной и автодекрементной адресации. Они называются преиндексным режимом с обратной записью и постиндексным режимом. Определения всех трех режимов адресации приведены ниже.

- *Преиндексный режим* — исполнительный адрес операнда представляет собой сумму содержимого базового регистра R*n* и величины смещения.
- *Преиндексный режим с обратной записью* — исполнительный адрес операнда генерируется так же, как в преиндексном режиме, после чего записывается обратно в регистр R*n*.
- *Постиндексный режим* — исполнительным адресом операнда является содержимое регистра R*n*.

Синтаксис языка ассемблера для всех трех режимов адресации, а также выражения для вычисления исполнительного адреса (EA) и код операции обратной записи приведены в табл. 3.1. Восклицательный знак обозначает обратную запись в преиндексном режиме адресации. В постиндексном режиме всегда выполняется обратная запись, поэтому восклицательный знак здесь не нужен. Обратите внимание, что в преиндексном и постиндексном режимах квадратные скобки расставлены по-разному. Если в скобки заключен только базовый регистр, его содержимое используется в качестве исполнительного адреса. Информация о смещении добавляется к содержимому регистра уже после доступа к операнду, то есть выполняется постиндексация. Это обобщенная форма автоинкрементного режима адресации, описанного в разделе 2.5. Когда в квадратные скобки заключены и базовый регистр и смещение, исполнительным адресом операнда является их сумма, то есть используется преиндексный режим. Если должна быть выполнена обратная запись, то на это указывает восклицательный знак. Преиндексный режим с обратной записью является обобщенным вариантом автодекрементного режима адресации, описанного в разделе 2.5.

Таблица 3.1. Индексные режимы адресации процессора ARM

Адресация	Синтаксис языка ассемблера	Формирование адреса
С непосредственно заданным смещением		
Преиндексация	[R*n*, #смещение]	EA=[R*n*] + смещение
Преиндексация с обратной записью	[R*n*, #смещение]!	EA=[R*n*] + смещение; R*n* ← [R*n*] + смещение
Постиндексация	[R*n*], #смещение	EA=[R*n*]; R*n* ← [R*n*] + смещение
Со значением смещения в регистре R*m*		
Преиндексация	[R*n*, ± R*m*, сдвиг]	EA=[R*n*] ± [R*m*] со сдвигом
Преиндексация с обратной записью	[R*n*, ± R*m*, сдвиг]!	EA=[R*n*] ± [R*m*] со сдвигом; R*n* ← [R*n*] ± [R*m*] со сдвигом
Постиндексация	[R*n*], ± R*m*, сдвиг	EA=[R*n*]; R*n* ← [R*n*] ± [R*m*] со сдвигом
Относительная (Преиндексация с непосредственно заданным смещением)	Адрес	EA=Адрес =[PC] + смещение

EA — исполнительный адрес (effective address).

Смещение — число со знаком, заданное в команде.

Сдвиг = направление # целое_число, где направление принимает значение LSL (сдвиг влево) или LSR (сдвиг вправо), а целое_число — это 5-битовое беззнаковое число, определяющее величину сдвига.

±R*m* — значение смещения, хранящееся в регистре R*m*, которое может быть прибавлено или отнято от содержимого базового регистра R*n*.

Во всех трех режимах адресации смещение может быть задано непосредственно в команде как значение из диапазона ±4095. В качестве альтернативы это можно сделать в регистре R*m*, указав знак (направление) смещения, в виде префикса ±, перед именем регистра. Например, команда

```
LDR  R0, [R1, -R2]!
```

выполняет операцию

$$R0 \leftarrow [[R1] - [R2]]$$

Исполнительный адрес операнда этой команды, [R1] − [R2], загружается в регистр R1, поскольку восклицательный знак обозначает обратную запись.

Когда смещение задается в регистре, оно умножается на степень двойки путем сдвига. Сдвиг в языке ассемблера задается при помощи значения LSL (сдвиг влево) или LSR (сдвиг вправо). Данное значение и указанная после него величина смещения следуют за именем регистра, R*m*, как показано в табл. 3.1. Величина смещения

задается как целое число из диапазона от 0 до 31. Например, перед использованием в качестве смещения содержимое регистра R2 в приведенном выше примере может быть умножено на 16 следующим образом:

```
LDR  R0, [R1, −R2,LSL #4]!
```

Данная команда выполнит такую операцию:

$$R0 \leftarrow [[R1] - 16 \times [R2]]$$

а затем загрузит исполнительный адрес в регистр R1.

В качестве базового регистра R*n* может использоваться счетчик команд, PC. Этот случай соответствует относительному режиму адресации, описанному в разделе 2.5. Ассемблер определяет непосредственно заданное смещение как расстояние со знаком между адресом операнда и содержимым обновленного регистра PC. Если исполнительный адрес вычисляется во время выполнения программы, в регистр PC записывается адрес слова, расположенного на два машинных слова дальше текущей команды (в которой указывается относительный режим адресации). Связано это с конвейерным выполнением команд, о котором рассказывается в главе 8.

На рис. 3.3, *а* приведен пример относительного режима адресации. Адрес операнда, символически заданный в команде как ITEM, равен 1060. Архитектура ARM не предусматривает абсолютной адресации. Поэтому, когда в языке ассемблера адрес задается так, как показано на этом рисунке, всегда используется относительная адресация. Она реализуется как преиндексная адресация с непосредственно заданным смещением и с указанием регистра PC в качестве базового. Как следует из данного рисунка, вычисленное ассемблером смещение равно 52, поскольку в момент его определения в процессе выполнения программы обновленный регистр PC содержит значение 1008, а исполнительный адрес, который должен быть сгенерирован, равен 1060 = 1008 + 52. Операнд должен находиться не более чем на 4095 байт выше или ниже адреса, содержащегося в обновленном регистре PC. Если заданный в команде адрес операнда находится вне этого диапазона, ассемблер сообщает об ошибке, и в таком случае должен использоваться другой режим адресации.

На рис. 3.3, *б* дан пример преиндексного режима адресации со смещением, указанным в регистре R6, и базовым значением, указанным в регистре R5. Команда Store (STR) сохраняет содержимое регистра R3 в слове памяти по адресу 1200.

Представленные на рис. 3.4 примеры наглядно демонстрируют, для чего нужна обратная запись в постиндексном и преиндексном режимах адресации. На рис. 3.4, *а* показаны первые три числа списка из 25 чисел, хранящихся в памяти начиная с адреса 100 и отстоящих на 25 слов друг от друга. Они составляют первую строку матрицы чисел 25 × 25, располагающуюся в памяти по столбцам. Первый столбец первой строки матрицы хранится по адресу 1000. Числа с адресами 1100, 1200, ..., 3400 составляют первую строку матрицы, а 25 чисел с адресами 1000, 1004, 1008, ..., 1096 — ее первый столбец.

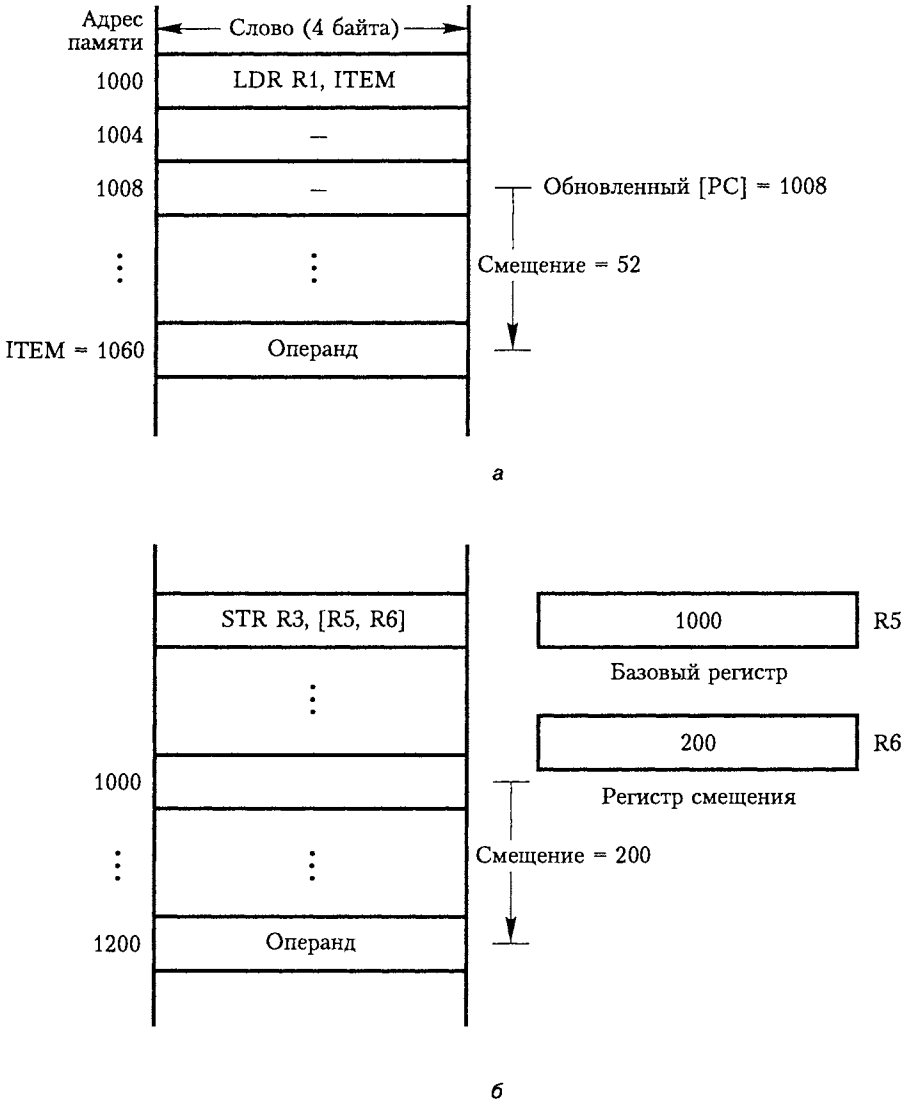


Рис. 3.3. Примеры режимов адресации процессора ARM: относительный (*а*), преиндексный (*б*)

Для доступа к последовательным числам первой строки матрицы удобнее всего воспользоваться постиндексной адресацией с обратной записью и смещением в регистре. Предположим, что базовым регистром служит R2 и что в нем содержится значение начального адреса 1000. Регистр R10 предназначен для хранения смещения, и в него загружается значение 25. Далее может быть выполнен программный цикл, на каждом шаге которого команда

```
LDR  R1,[R2],R10,LSL #2
```

будет загружать в регистр R1 очередной элемент первой строки матрицы. А теперь давайте рассмотрим эту программу более подробно. Когда команда загрузки выполняется в первый раз, исполнительный адрес [R2] равен 1000. Расположенное по данному адресу число 6 помещается в регистр R1. Далее операция обратной записи изменяет содержимое регистра R2 с 1000 на 1100, для того чтобы он указывал на следующее число, −17. Для этого содержимое регистра смещения R10, равное 25, сдвигается на два разряда влево и результат прибавляется к содержимому регистра R2. Содержимое регистра R10 при этом не меняется. Сдвиг влево эквивалентен умножению числа 25 на 4, так что в результате генерируется смещение 100. После добавления значения смещения к содержимому регистра R2 в этот регистр записывается новый адрес — 1100. И когда команда загрузки выполняется повторно, на второй итерации цикла, в регистр R1 записывается второе число из хранящихся в матрице, то есть −17. Третье число, 321, загружается в R1 на третьей итерации и т. д.

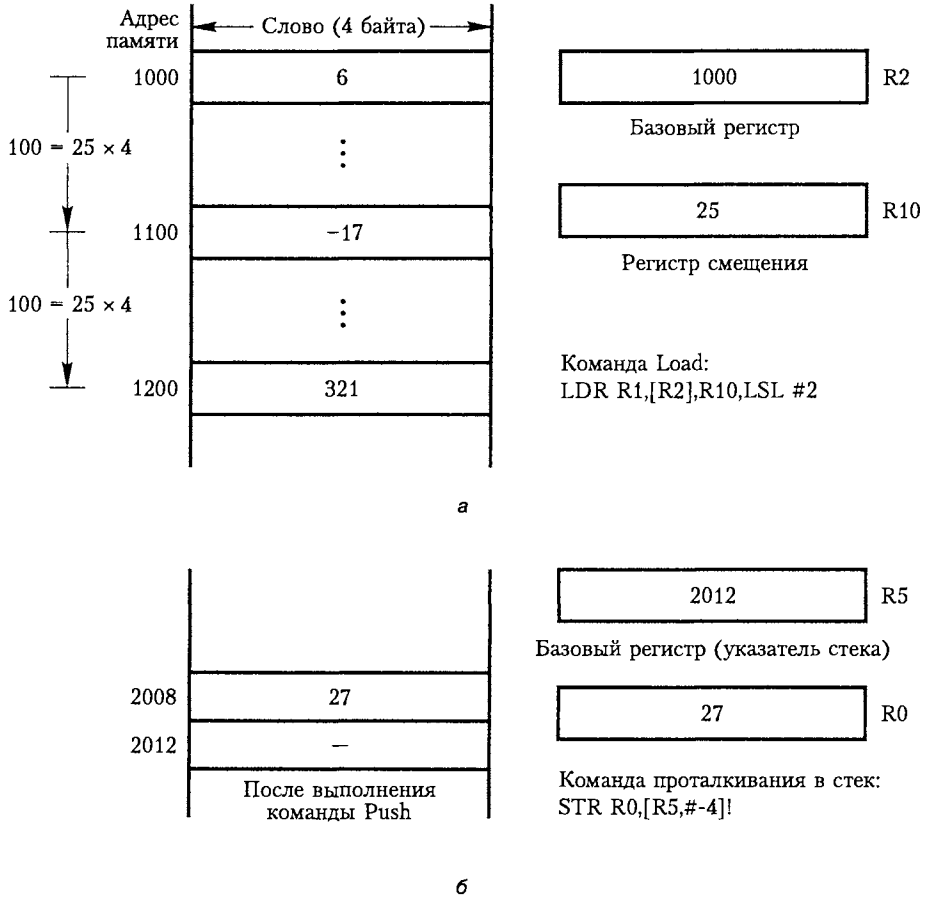


Рис. 3.4. Режимы адресации процессора ARM, в которых используется обратная запись: постиндексный (*а*); преиндексный (*б*)

В этом примере к содержимому базового регистра добавляется сдвинутое содержимое регистра смещения. Как следует из табл. 3.1, его можно не только добавлять, но и вычитать из содержимого регистра смещения. Допускается сдвиг вправо или влево на любое количество разрядов от 0 до 31.

На рис. 3.4, *б* приведен пример проталкивания в стек содержимого регистра R0 (числа 27). Регистр R5 используется в качестве указателя стека. Первоначально он содержит число 2012 — адрес текущей вершины стека. Воспользовавшись преиндексной адресацией с обратной записью, операцию помещения в стек можно выполнить посредством такой команды:

```
STR  R0,[R5,#-4]!
```

Непосредственно заданное смещение −4 добавляется к содержимому регистра R5 (числу 2012), а результат опять записывается в R5. Новый адрес вершины стека, 2008, используется как исполнительный адрес операции сохранения. Эта операция, определяемая командой STR, записывает по адресу 2008 содержимое регистра R0 (число 27).

Загрузка и сохранение нескольких операндов

В дополнение к командам загрузки и сохранения, имеющим по одному операнду, процессор ARM поддерживает еще две команды, которые могут загрузить и сохранить несколько операндов. Они называются командами *блочной пересылки* и предназначены для загрузки из памяти или сохранения в памяти любого подмножества регистров общего назначения. Допускается применение только операндов длиной в одно слово. Указанные команды называются LDM (Load Multiple — множественная загрузка) и STM (Store Multiple — множественное сохранение). В памяти операнды должны располагаться в последовательных словах. Разрешаются все формы преиндексной и постиндексной адресации с обратной записью и без таковой. Они относятся к заданному в команде базовому регистру R*n*. Смещение, производимое этими командами, всегда равно 4, поэтому задавать его явно не нужно. Список регистров должен быть создан в порядке возрастания их номеров в виде выражения языка ассемблера. В качестве примера рассмотрим случай, когда базовым регистром является R0, который первоначально содержит значение 1000. Тогда команда

```
LDMIA  R10!,{R0,R1,R6,R7}
```

перешлет слова, расположенные в памяти по адресам 1000, 1004, 1008, 1012 в регистры R0, R1, R6, R7, и после выполнения последней операции пересылки в регистре R10 останется адрес 1016. Суффикс IA в коде операции означает «Increment After» (постинкрементация) и соответствует постиндексному режиму адресации. В разделе 3.6 мы еще вернемся к командам множественной загрузки и сохранения и поговорим о них более подробно при обсуждении подпрограмм, где они используются для сохранения в стеке значений регистров и восстановления их из стека.

3.1.3. Регистровые команды пересылки

Часто программисту приходится копировать содержимое одного регистра в другой или загружать некоторое непосредственно заданное значение. Команда пересылки

```
MOV  Rd,Rm
```

формат которой вы видите на рис. 3.2, копирует содержимое регистра R*m* в регистр R*d*. С помощью этой же команды можно загрузить в регистр R*d* операнд, заданный в ее 8 младших разрядах. Так, команда

```
MOV  R0,#76
```

загружает в регистр R0 значение 76. В обеих формах команды пересылки исходный операнд, прежде чем помещать его в регистр назначения, можно сдвинуть.

3.2. Арифметические и логические команды

Процессор ARM поддерживает множество команд, предназначенных для выполнения арифметических и логических операций над операндами, которые либо содержатся в регистре общего назначения, либо задаются непосредственно в команде. В частности, имеются команды для разных способов сложения и вычитания, две команды для умножения, а также для выполнения логических операций И, ИЛИ, НЕ, Исключающее ИЛИ и очистки бита (bit clear). А еще существуют команды сравнения, устанавливающие флаги кодов условий по результатам арифметических или логических операций с двумя операндами. Они не записывают в регистры результаты выполняемых операций. Большинство таких команд имеют формат, показанный на рис. 3.2.

3.2.1. Арифметические команды

Базовая структура арифметических команд в языке ассемблера такова:

```
КодОперации  Rd,Rm,Rn
```

Подобная команда выполняет операцию над операндами, хранящимися в регистрах общего назначения R*m* и R*n*, и помещает результат в регистр R*d*. В частности, команда

```
ADD  R0,R2,R4
```

выполняет операцию

```
R0 ← [R2] + [R4]
```

а команда

```
SUB  R0,R6,R5
```

выполняет операцию

```
R0 ← [R6] − [R5]
```

Вместо того чтобы помещать второй операнд в регистр R*m*, можно задать его прямо в команде. Так, команда

```
ADD  R0,R3,#17
```

выполняет операцию

```
R0 ← [R3] + 17
```

Непосредственно заданное значение содержится в 8-разрядном поле в битах команды b_{7-0}.

В команде, до ее выполнения, можно произвести сдвиг или циклический сдвиг второго операнда. Операция сдвига задается в конце команды. Несколько слов о том, как работает, скажем, команда

ADD R0,R1,R5,LSL #6

Сначала второй операнд, содержащийся в регистре R5, сдвигается влево на 4 разряда (что эквивалентно операции [R5] × 16), затем результат прибавляется к содержимому регистра R1 и сумма помещается в регистр R0.

Существует две версии команды умножения. Первая из них перемножает содержимое двух регистров и помещает 32 младших разряда полученного произведения в третий регистр. Старшие биты произведения, если таковые имеются, просто игнорируются. Например, команда

MUL R0,R1,R2

выполняет операцию

R0 ← [R1] × [R2]

Во второй версии команды умножения задается четвертый регистр, содержимое которого прибавляется к произведению перед сохранением результата в регистре назначения. Таким образом, команда

MLA R0,R1,R2,R3

выполняет операцию

R0 ← [R1] × [R2] + [R3]

Такая операция называется умножением с накоплением. Она часто используется в алгоритмах обработки цифровых сигналов. Примеры приложений этого типа приведены в разделе 3.7. Четвертый регистр задается в поле «Другая информация», показанном на рис. 3.2. В командах умножения не предусмотрена возможность сдвига или циклического сдвига операндов перед их использованием. Некоторые версии архитектуры системы команд ARM поддерживают операцию умножения для чисел двойной длины (см. главу 11).

Операции сдвига операндов

Ранее в этой главе уже было отмечено, что одной из отличительных особенностей системы команд процессора ARM является то, что все эти команды выполняются условно. Еще одной ее особенностью принято считать интеграцию в большинство команд операций сдвига и циклического сдвига операндов. В системах команд других компьютеров операции сдвига обычно выполняются отдельными командами. В частности, такие команды применяются процессорами Motorola 68000 и Intel IA-32, описанными во второй и третьей частях настоящей главы. Интеграция в команды операций сдвига и циклического сдвига позволяет уменьшить размер кода и ускорить его выполнение. Она реализуется при помощи специальной

схемы циклического сдвига, размещенной между регистрами и арифметико-логическим устройством в процессоре. Подробное описание операций сдвига и циклического сдвига процессора ARM приведено в приложении Б.

3.2.2. Логические команды

Логические операции И, ИЛИ, Исключающее ИЛИ и очистки бита в архитектуре ARM реализованы в виде команд AND, ORR, EOR и BIC. У них тот же формат, что и у арифметических команд. Например, команда

AND R*d*, R*n*,R*m*

выполняет операцию

[R*d*] ← [R*n*] ∧ [R*m*]

то есть поразрядное логическое И над операндами, хранящимися в регистрах R*n* и R*m*. Если в регистре R0 содержится, предположим, шестнадцатеричное значение 02FA62CA, а в регистре R1 — значение 0000FFFF, то команда

AND R0,R0,R1

поместит в регистр R0 значение 000062CA.

Команда BIC тесно связана с командой AND. Она дополняет каждый бит операнда R*m*, а затем выполняет поразрядную операцию AND над результатом дополнения и содержимым регистра R*n*. Если использовать те же значения регистров R0 и R1, что и в предыдущем примере, команда

AND R0,R0,R1

поместит в регистр R0 значение 02FA0000.

Команда Move Negative с кодом операции MVN дополняет биты исходного операнда и помещает результат в регистр R*d*. Если содержимым регистра R3 является шестнадцатеричное значение 0F0F0F0F, то команда

MVN R0,R3

помещает в регистр R0 значение F0F0F0F0.

Программа упаковки цифр

На рис. 3.5 приведена программа для упаковки двух десятичных цифр в байт памяти. Код универсальной версии этой программы был приведен на рис. 2.31 и описан в разделе 2.10.2. Десятичные цифры, представленные в коде ASCII, хранятся в байтах по адресам LOC и LOC + 1. Программа упаковывает соответствующие 4-битовые коды BCD в один байт по адресу PACKED.

В первой команде загрузки представленной на рис. 3.5 программы предполагается, что адрес LOC хранится в памяти по адресу POINTER. Как вы узнаете из раздела 3.4, для помещения адреса LOC в память по адресу POINTER можно воспользоваться специальной директивой ассемблера. Мы применяем такой способ

загрузки адреса LOC в регистр R0 потому, что 32-разрядные адреса не могут непосредственно задаваться в командах. Значение по адресу POINTER указывает на два ASCII-кода цифр, хранящихся в последовательных байтах. Эти коды, четыре младших бита которых представляют собой BCD-коды тех же цифр, следующими двумя командами загрузки записываются в младшие байты регистров R1 и R2. Далее команда AND очищает 28 старших разрядов регистра R0, заполняя их нулями и оставляя в четырех младших разрядах BCD-код цифры. Затем команда ORR сдвигает первую BCD-цифру на четыре разряда влево и помещает ее слева от второй BCD-цифры в регистре R2. В завершение упакованные цифры из младшего байта регистра R2 сохраняются в памяти по адресу PACKED.

```
LDR     R0,POINTER          Загрузка адреса LOC в R0
LDRB    R1,[R0]             Загрузка символов ASCII
LDRB    R2,[R0,#1]          в регистры R1 и R2
AND     R2,R2,#&F           Очистка 28 старших разрядов регистра R2
ORR     R2,R2,R1,LSL, #4    Добавление в регистр R2 сдвинутого влево
                            содержимого регистра R1
STRB    R2,PACKED           Сохранение упакованных цифр BCD в памяти
                            по адресу PACKED
```

Рис. 3.5. Программа на языке процессора ARM для упаковки 4-битовых десятичных цифр в один байт

3.3. Команды перехода

В командах условного перехода задается 24-разрядное смещение в форме дополнения до двух со знаком, которое должно быть добавлено к обновленному содержимому регистра PC, в результате чего получится целевой адрес перехода. Формат команды перехода приведен на рис. 3.6, *а*, а ее пример — на рис. 3.6, *б*. Команда BEQ (Branch if Equal to 0 — переход, если равно 0) вызывает переход, если флаг Z установлен в 1.

Условие, от которого зависит направление перехода, задается в четырех старших разрядах слова команды, b_{31-28}. Команда перехода подобно любой другой команде ARM выполняется лишь в том случае, если текущее состояние флагов кодов условий соответствует условию, заданному в поле «Условие» команды.

К моменту вычисления целевого адреса перехода содержимое регистра PC уже обновлено и включает адрес команды, расположенной на два слова дальше команды перехода. Например, если команда перехода располагается по адресу 1000, а целевой адрес равен 1100 (как на рис. 3.6, *б*), то величина смещения должна быть равной 92, поскольку в момент вычисления адреса 1100 обновленный регистр PC будет содержать значение 1000 + 8 = 1008.

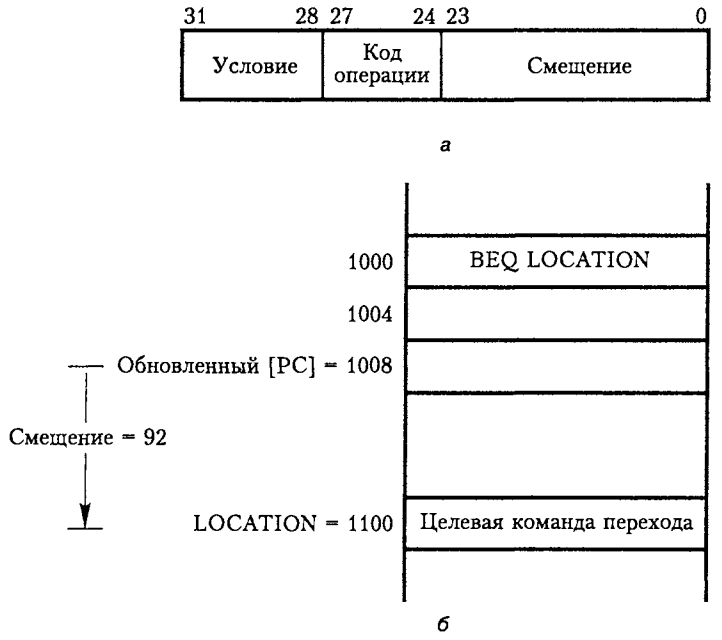


Рис. 3.6. Команды перехода процессора ARM: формат команды (*а*); определение целевого адреса перехода (*б*)

3.3.1. Установка кодов условий

Единственным назначением некоторых команд, в том числе и команды сравнения

CMP Rn,Rm

выполняющей операцию

[Rn] – [Rm]

является установка флагов кодов условий на основе результата операции вычитания. С другой стороны, арифметические и логические операции воздействуют на флаги кодов условий только в том случае, если это явно указано в соответствующем бите кода операции. В ассемблерном коде операции установка флагов обозначается суффиксом S. Так, команда

ADDS R0,R1,R2

устанавливает флаги кодов условий, а команда

ADD R0,R1,R2

не устанавливает таковые.

3.3.2. Программа с циклом для сложения чисел

В программе, которую мы рассмотрим далее, для сложения нескольких чисел применяется цикл (рис. 3.7). В операциях загрузки и сохранения, выполняемых

первой, второй и последней командами, используется относительная адресация. При этом предполагается, что адреса N, POINTER и SUM находятся в допустимом диапазоне смещения относительно адреса, заданного в регистре PC. В памяти по адресу POINTER содержится адрес NUM1 первого из складываемых чисел, по адресу N — количество чисел в списке, а по адресу SUM должна храниться результирующая сумма. Постиндексный режим с обратной записью, используемый в первой команде цикла, соответствует автоинкрементному режиму, применявшемуся в программе, представленной на рис. 2.16.

```
        LDR   R1,N          Загрузка количества элементов списка в R1
        LDR   R2,POINTER    Загрузка адреса NUM1 в R2
        MOV   R0,#0         Очистка сумматора R0
LOOP    LDR   R3,[R2],#4    Загрузка следующего числа в R3
        ADD   R0,R0,R3      Сложение чисел и помещение результата в R0
        SUBS  R1,R1,#1      Декремент счетчика цикла R1
        BGT   LOOP          Переход на начало цикла, если сложены не все
                            числа
        STR   R0,SUM        Запись суммы в память
```

Рис. 3.7. Программа для процессора ARM, выполняющая сложение последовательности чисел

3.4. Язык ассемблера

В языке ассемблера процессора ARM имеются директивы, задающие порядок и методы резервирования памяти, присвоения числовых значений меткам адресов и символьным константам, определения адресов памяти, по которым должны размещаться программа и блоки данных, а также для определения конца исходного текста программы.

Примеры использования некоторых из этих директив вы видите на рис. 3.8, в программе, представляющей собой полный текст программы, основная часть которой приведена на рис. 3.7. Директива AREA с аргументом CODE или DATA указывает на начало блока памяти, в котором содержатся команды программы или данные. Обычно в ней задаются и другие параметры, но их мы обсуждать не будем. Директива ENTRY определяет, что выполнение программы должно начаться со следующей за ней команды — в нашем случае команды LDR.

В области данных, которая следует за областью кода, с помощью директив DCD определяются значения, используемые в качестве операндов команд программы. Эти директивы объявляют метки и присваивают им значения. Первые две директивы DCD инициализируют значениями 0 и 5 слова памяти с метками SUM и N. Следующая директива помещает по адресу POINTER адрес NUM1. А последняя директива DCD определяет пять чисел, которые должны быть последовательно помещены в память начиная с адреса NUM1.

В шестнадцатеричных значениях констант используется префикс &, а константы по основанию *n*, где *n* лежит в диапазоне от 2 до 9, обозначаются как *n_xxx*. Например, двоичная константа может быть представлена так: 2_101100. Константы по основанию 10 задаются без префикса.

	Метка адреса памяти	Операция	Адресная информация и данные
Директивы ассемблера		AREA	CODE
		ENTRY	
Команды, на основе которых генерируются машинные команды		LDR	R1,N
		LDR	R2,POINTER
		MOV	R0,#0
	LOOP	LDR	R3,[R2],#4
		ADD	R0,R0,R3
		SUBS	R1,R1,#1
		BGT	LOOP
		STR	R0,SUM
Директивы ассемблера		AREA	DATA
	SUM	DCD	0
	N	DCD	5
	POINTER	DCD	NUM1
	NUM1	DCD	3,-17,27,-12,322
		END	

Рис. 3.8. Более полный вариант программы на языке ассемблера, приведенной на рис. 3.7

Для определения символических имен констант используется директива EQU. В частности, команда

```
TEN EQU 10
```

позволяет вместо десятичной константы 10 использовать в программе символическое имя TEN. Когда в программе задействуется большое количество регистров, для них удобно использовать символические имена, соответствующие их назначению. Это делается с помощью директивы RN. Например, команда

```
COUNTER RN 3
```

назначает регистру R3 имя COUNTER. В ассемблере предопределены имена регистров R0–R15, PC (для R15) и LR (для R14).

3.4.1. Псевдокоманды

В языке ассемблера имеется альтернативный способ загрузки адреса NUM1 в регистр R2 (рис. 3.8). Так, псевдокоманда

```
ADR Rd,ADDRESS
```

загружает 32-разрядное значение ADDRESS в регистр R*d*. Это не настоящая машинная команда. Для реализации псевдокоманд ассемблер сам выбирает подходящие машинные команды. Одним из способов реализации псевдокоманды

```
ADR R2,NUM1
```

является использовавшаяся на рис. 3.8 комбинация команды

```
LDR R2,POINTER
```

и директивы

```
POINTER DCD NUM1
```

Эти две команды в программе можно было бы заменить одной псевдокомандой, поместив ее на место команды LDR. В таком случае ассемблеру самому нужно будет выделить область данных для объявления DCD.

В нашем примере существует более эффективный способ реализации команды ADR, который и будет выбран ассемблером. Если загружаемый командой ADR адрес более чем на 255 байт отличается от текущего содержимого регистра PC (R15), указанная псевдокоманда может быть реализована командой

```
ADD Rd,R15,#смещение
```

При использовании этой команды в нашем примере адрес POINTER вообще не понадобится. Ассемблер реализует псевдокоманду ADR при помощи машинной команды

```
ADD R2,R15,#28
```

поскольку в момент выполнения команды ADD адрес NUM1 располагается на 28 байт дальше адреса, содержащегося в обновленном регистре PC. При этом предполагается, что область данных начинается сразу за командой STR. На самом деле это не так, поскольку за командой STR должна следовать команда, возвращающая управление операционной системе, но мы ее опустили.

3.5. Команды ввода-вывода

В архитектуре процессоров ARM применяются команды ввода-вывода с отображением в память, о чем было рассказано в разделе 2.7. Это означает, что для чтения символа с клавиатуры или его вывода на дисплей можно использовать программно-управляемый ввод-вывод.

Предположим, что в регистрах состояния устройств INSTATUS (клавиатура) и OUTSTATUS (дисплей) содержатся соответственно управляющие флаги SIN и SOUT. Регистры клавиатуры DATAIN и дисплея DATAOUT располагаются по адресам INSTATUS + 4 и OUTSTATUS + 4, сразу за регистрами состояния. Циклы ожидания операций чтения и записи можно реализовать следующим образом. Предположим, что адрес INSTATUS загружен в регистр R1. В таком случае приведенная далее последовательность команд считывает в регистр R3 введенный с клавиатуры символ:

```
READWAIT  LDR   R3,[R1]
          TST   R3,#8
          BEQ   READWAIT
          LDRB  R3,[R1,#4]
```

Команда TST выполняет над своими двумя операндами поразрядную логическую операцию AND и с учетом полученного результата устанавливает соответствующие флаги условий. В непосредственно заданном операнде 8 установлен единственный бит — в третьем разряде. Поэтому результат операции TST будет равен нулю, если в третьем разряде регистра INSTATUS содержится нуль, и не будет равен нулю в том случае, если в указанном разряде содержится единица, обозначающая, что в регистре DATAIN имеется очередной введенный пользователем символ. Команда BEQ осуществляет переход по метке READWAIT, если результат предшествующего сравнения равен нулю. В итоге получается цикл, выполняемый до нажатия клавиши, в ответ на которое третий разряд регистра INSTATUS устанавливается в 1.

Если предположить, что в регистр R2 загружен адрес регистра OUTSTATUS, команды

```
WRITEWAIT  LDR   R4,[R2]
           TST   R4,#8
           BEQ   WRITEWAIT
           STRB  R3,[R2,#4]+
```

перешлют символ из регистра R3 в регистр DATAOUT, когда дисплей будет готов его принять.

Эти две подпрограммы можно использовать для чтения с клавиатуры строки символов, сохранения ее в памяти и вывода на экран, как это делалось в программе, представленной на рис. 3.9. Данная программа написана на основе программы, приведенной на рис. 2.20. Предполагается, что в регистре R0 содержится адрес первого байта области памяти, в которой будет храниться строка. Регистры с R1 по R4 используются так же, как в описанных выше циклах READWAIT и WRITEWAIT. Первая команда сохранения (STRB) помещает считанный с клавиатуры символ в память. В ней используется постиндексный режим адресации, с помощью которого выполняется проход по области памяти, — так же, как при помощи автоинкрементного режима адресации, задействованного в упоминаемой программе из главы 2. Команда TEQ (Test if Equal — проверка, если равно) проверяет, равны ли два операнда, и соответствующим образом устанавливает флаг кода условия Z.

```
READ   LDR   R3,[R1]     Загрузка [INSTATUS]
       TST   R3,#8       и ожидание символа
       BEQ   READ
       LDRB  R3,[R1,#4]  Чтение символа и его
       STRB  R3,[R0],#1  сохранение в памяти
ECHO   LDR   R4,[R2]     Загрузка [OUTSTATUS]
       TST   R4,#8       и ожидание готовности дисплея
       BEQ   ECHO
       STRB  R3,[R2,#4]  Отправка символа на экран
       TEQ   R3,#CR      Если не вводится символ возврата каретки,
       BNE   READ        считывается следующий символ
```

Рис. 3.9. Программа для процессора ARM, которая считывает символы с клавиатуры и выводит их на экран

3.6. Подпрограммы

Для вызова подпрограммы предназначена команда BL (Branch and Link — переход с возвратом). Она действует так же, как любая другая команда перехода, и имеет один дополнительный шаг. Адрес возврата, который является адресом следующей за BL команды, загружается в регистр R14, выполняющий функции регистра связи. Поскольку подпрограммы могут быть вложенными, содержимое регистра связи может быть сохранено подпрограммой в стеке. В качестве указателя на стек обычно используется регистр R13.

Обратимся к рис. 3.10. Здесь приведена та же программа, что и на рис. 3.7, но оформленная в виде подпрограммы. Параметры ей передаются через регистры. Вызывающая программа передает подпрограмме размер списка чисел и адрес его первого элемента в регистрах R1 и R2, а подпрограмма возвращает ей сумму в регистре R0. Кроме того, подпрограмма использует регистр R3. Поэтому его содержимое вместе с содержимым регистра связи R14 сохраняется в стеке командой STMFD. Суффикс FD в этой команде указывает на то, что стек увеличивается в направлении уменьшения адресов и что указатель стека R13 перед проталкиванием очередного элемента должен уменьшаться. Команда LDMFD восстанавливает содержимое регистра R3, а затем проталкивает сохраненный адрес возврата в регистр PC (R15), автоматически выполняя операцию возврата.

```
Вызывающая программа
                LDR     R1,N
                LDR     R2,POINTER
                BL      LISTADD
                STR     R0,SUM
                :
Подпрограмма
LISTADD         STMFD   R13!,{R3,R14}   Сохранение R3 и адреса возврата из R14
                                        в стеке с использованием R13 как
                                        указателя стека
                MOV     R0,#0
                LDR     R3,[R2],#4
                ADD     R0,R0,R3
                SUBS    R1,R1,#1
                BGT     LOOP
                LDMFD   R13!,{R3,R15}   Восстановление R3 и загрузка адреса
                                        возврата в PC (R15)
```

Рис. 3.10. Программа с рис. 3.7, реализованная в виде подпрограммы, в которой параметры передаются через регистры

На рис. 3.11, *а* показана та же программа, что и на рис. 3.7, оформленная, опять-таки, в виде подпрограммы, правда, в данном случае параметры передаются через стек. Первые четыре команды вызывающей программы проталкивают в стек параметры NUM1 и *n*. Мы предполагаем, что значение NUM1 содержится в памяти по адресу POINTER. Регистры R0–R3 используются так же, как в подпрограмме с рис. 3.7. Их содержимое сохраняется в стеке первой командой подпрограммы вместе с адресом возврата, хранящимся в регистре R14. На рис. 3.11, *б* приведено

содержимое стека в разные моменты работы программы. После помещения параметров в стек и выполнения команды вызова подпрограммы BL вершина стека располагается на уровне 2, а после сохранения всех регистров первой командой подпрограммы она оказывается на уровне 3. Следующие две команды загружают параметры из стека в регистры R1 и R2. Смещение значений *n* и NUM1, равное 20 и 24 байтам, задается относительно текущей вершины стека (то есть относительно уровня 3). Вычисленная в регистре R0 сумма значений сразу же помещается командой STR в стек на место параметра NUM1.

(Вершина стека располагается на уровне 1, как показано ниже.)

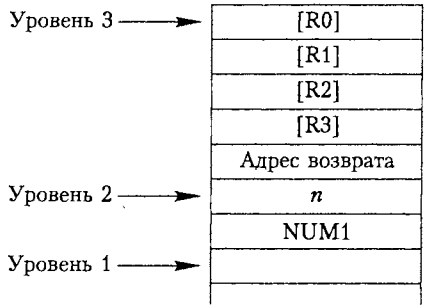
Вызывающая программа

```
            LDR     R0,POINTER      Проталкивание NUM1
            STR     R0,[R13,#-4]!   в стек
            LDR     R0,N            Проталкивание значения n
            STR     R0,[R13,#-4]!   в стек
            BL      LISTADD
            LDR     R0,[R13,#4]     Сохранение суммы в памяти по адресу
                                    SUM
            STR     R0,SUM
            ADD     R13,R13,#8      Удаление параметров из стека
            :
```

Подпрограмма

```
LISTADD     STMFD   R13!,{R0-R3,R14}  Сохранение регистров
            LDR     R1,[R13,#20]      Загрузка параметров из стека
            LDR     R2,[R13,#24]
            MOV     R0,#0
LOOP        LDR     R3,[R2],#4
            ADD     R0,R0,R3
            SUBS    R1,R1,#1
            BGT     LOOP
            STR     R0,[R13,#24]      Помещение суммы в стек
            LDMFD   R13!,{R0-R3,R15}  Восстановление регистров и возврат
```

а



б

Рис. 3.11. Программа с рис. 3.7, реализованная в виде подпрограммы, в которой параметры передаются через стек: главная программа и подпрограмма (*а*); вершина стека в разные моменты времени (*б*)

Адрес памяти				Комментарии
Вызывающая программа (Main)				
		⋮		
2000		LDR	R0,PARAM2	Помещение параметров в стек
2004		STR	R10,[SP,#−4]!	
2008		LDR	R10,PARAM1	
2012		STR	R10,[SP,#−4]!	
2016		BL	SUB1	
2020		LDR	R10,[SP]	Сохранение результата выполнения подпрограммы SUB1
2024		STR	R10,RESULT	
2028		ADD	SP,SP,#8	Удаление параметров из стека
2032		Следующая команда		
		⋮		
Первая подпрограмма				
2100	SUB1	STMFD	SP!,{R0−R3,FP,LR}	Сохранение регистров
2014		ADD	FP,SP,#16	Загрузка указателя фрейма
2108		LDR	R0,[FP,#8]	Загрузка параметров
2112		LDR	R1,[FP,#12]	
		⋮		
		LDR	R2,PARAM3	Помещение параметров в стек
		STR	R2,[SP,#−4]!	
2160		BL	SUB2	
2164		LDR	R2,[SP],#4	Выталкивание результата выполнения подпрограммы SUB2 из стека в R2
		⋮		
				Помещение результата в стек
		LDMFD	SP!,{R0−R3,FP,PC}	Восстановление регистров и возврат
Вторая подпрограмма				
3000	SUB2	STMFD	SP!,{R0,R1,FP,LR}	Сохранение регистров
		ADD	FP,SP,#8	Загрузка указателя фрейма
		LDR	R0,[FP,#8]	Загрузка параметра
		⋮		
		STR	R1,[FP,#8]	Помещение результата в стек
		LDMFD	SP!,{R0,R1,FP,PC}	Сохранение регистров и возврат

Рис. 3.12. Вложенные подпрограммы процессора ARM на языке ассемблера

Как видите, в приведенных подпрограммах используются вложенные вызовы. На рис. 3.12 представлена программа с рис. 2.28, переписанная для процессора ARM. На рис. 3.13 показаны стековые фреймы, соответствующие первой и второй подпрограммам. Указателем на фрейм служит регистр R12. Некоторым регистрам в этой программе присваиваются символические имена, облегчающие чтение кода.

В частности, регистры R12 (указатель на фрейм), R13 (указатель на стек), R14 (регистр связи) и R15 (счетчик команд) обозначены как FP, SP, LR и PC соответственно. Для задания этих имен можно воспользоваться ассемблерной директивой RN.

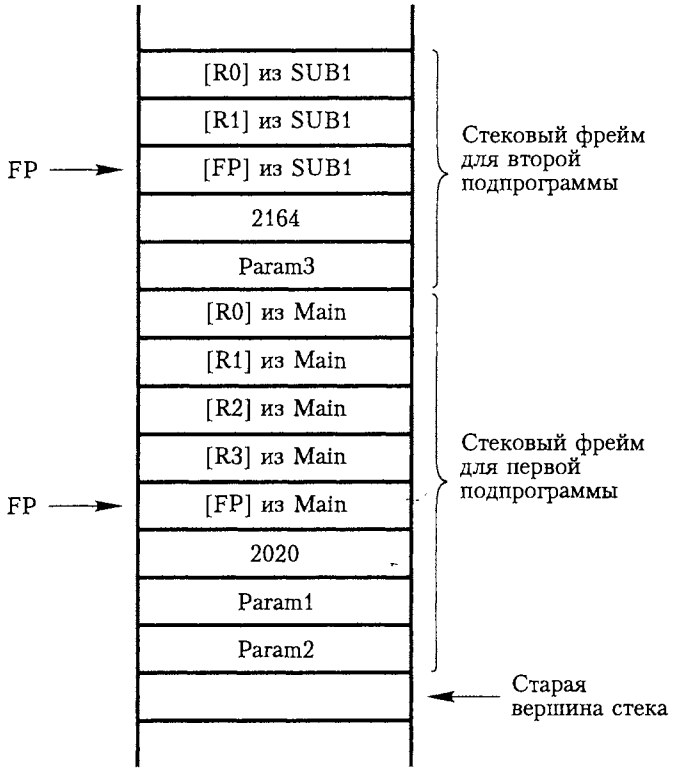


Рис. 3.13. Стековые фреймы программы, представленной на рис. 3.12

Структура вызывающей программы и подпрограмм осталась такой же, как на рис. 2.28. Для процессора ARM в программу внесены следующие изменения. Адрес возврата и старое содержимое стекового фрейма сохраняются в стеке первой командой каждой подпрограммы. Вторая команда связывает указатель стека с его же сохраненным значением, как показано на рис. 3.13. Это соответствует значениям указателя фрейма на рис. 2.27 и 2.29. Параметры, как обычно, располагаются со смещением 8, 12 и т. д. Последняя команда каждой подпрограммы восстанавливает старое значение указателя фрейма и значения других применяемых в подпрограмме регистров, а затем выталкивает из стека адрес возврата в регистр PC.

3.7. Примеры программ

В данном разделе рассматриваются ARM-версии программ для вычисления скалярного произведения, сортировки байтов и операций со связными списками,

описанных в главе 2. Эти программы созданы на основе программ, приведенных на рис. 2.33, 2.34, 2.37 и 2.38. Мы рассмотрим только те фрагменты ARM-кода, которые отличаются от кода, анализируемого в главе 2.

3.7.1. Программа для вычисления скалярного произведения двух векторов

Первые две команды на рис. 3.14 загружают в регистры R1 и R2 адреса векторов A и B. Это псевдокоманды ADR, описанные в разделе 3.4.1. Если адреса AVEC и BVEC расположены достаточно близко к командам программы, для их генерирования может быть применена команда Add с текущим значением регистра PC. Для доступа к содержимому N и DOTPROD используется относительный режим адресации, а для загрузки компонентов векторов в первых двух командах цикла — постиндексный режим адресации. Необходимые арифметические операции выполняет команда MLA (Multiply-Accumulate). Она перемножает элементы векторов, находящиеся в регистрах R4 и R5, и прибавляет результат к содержимому регистра R0.

```
        ADR   R1,AVEC      R1 указывает на вектор A
        ADR   R2,BVEC      R2 указывает на вектор B
        LDR   R3,N         R3 используется в качестве счетчика цикла
        MOV   R0,#0        В R0 накапливается сумма произведений
LOOP    LDR   R4,[R1],#4   Загрузка компонента вектора A
        LDR   R5,[R2],#4   Загрузка компонента вектора B
        MLA   R0,R4,R5,R0  Умножение компонентов и прибавление
                           произведения к содержимому регистра R0
        SUBS  R3,R3,#1     Уменьшение значения счетчика
        BNE   LOOP         Переход к началу цикла, если перемножены еще
                           не все компоненты
        STR   R0,DOTPROD   Сохранение скалярного произведения в памяти
```

Рис. 3.14. Программа для процессора ARM, вычисляющая скалярное произведение двух векторов

3.7.2. Программа сортировки байтов

На рис. 3.15 показана программа сортировки байтов. Она имеет ту же структуру, что и программа на рис. 2.34, *б*. Адрес первого байта массива LIST загружается в регистр R4. Он используется в предпоследней команде сравнения, определяющей на основе индекса k, когда должен завершиться внутренний цикл. Аналогичным образом, в регистре R5 содержится адрес LIST + 1, который применяется в последней команде сравнения, определяющей на основе индекса j, когда должен завершиться внешний цикл. Для уменьшения значения индекса j во внешнем цикле предназначен базовый регистр R2, а для уменьшения значения индекса k при проходе по каждому из подсписков во внутреннем цикле — базовый регистр R3. Для загрузки байтов LIST(j) в регистр R0 и байтов LIST(k) в регистр R1 во внешнем и внутреннем циклах используется преиндексный режим адресации.

```
int main(int argc, char* argv[])
{
   for (j = n-1; j>0; j = j-1)
   {for k = j-1; k>=0; k = k-1)
      {if (LIST[k] > LIST[j])
         {TEMP = LIST[k];
          LIST[k] = LIST[j];
          LIST[j] = TEMP;
         }
      }
   }
```

а

```
        ADR     R4,LIST     Загрузка указателя списка
        LDR     R10,N       в регистр R4 и инициализация базового
                            регистра внешнего цикла R2
        ADD     R2,R4,R10   значением LIST + n
        ADD     R5,R4,#1    Загрузка значения LIST + 1 в R5
OUTER   LDRB    R0,[R2,#-1]! Загрузка значения LIST(j) в R0
        MOV     R3,R2       Инициализация базового регистра внутреннего
                            цикла R3 значением LIST + n - 1
INNER   LDRB    R1,[R3,#-1]! Загрузка LIST(k) в R1
        CMP     R1,R0       Сопоставление LIST(k) и LIST(j)
        STRGTB  R1,[R2]     Если LIST(k) > LIST(j),
        STRGTB  R0,[R3]     LIST(k) и LIST(j) меняются местами, а в R0
        MOVGT   R0,R1       помещается новое значение LIST(j)
        CMP     R3,R4       Если k > 0, повторное выполнение
        BNE     INNER       внутреннего цикла
        CMP     R2,R5       Если j > 1, повторное выполнение
        BNE     OUTER       внешнего цикла
```

б

Рис. 3.15. Программа сортировки байтов для процессора ARM: на языке C (*а*); на языке ассемблера (*б*)

Чтобы поменять элементы LIST(*k*) и LIST(*j*) местами, производится условное выполнение команд, являющееся одной из отличительных особенностей системы команд процессора ARM. Последовательность из трех команд, STR, STR и MOV, реализуется лишь в том случае, если LIST(*k*) > LIST(*j*), на что указывает суффикс GT. Поэтому условный переход по метке NEXT, использовавшийся в программе на рис. 2.34, *б*, в программе для процессора ARM не нужен.

3.7.3. Подпрограммы для вставки и удаления элементов связного списка

Программы, которые выполняют вставку и удаление элементов связного списка (рис. 3.16 и 3.17), очень близки к программам, приведенным на рис. 2.37 и 2.38.

Однако в них не производятся условные переходы вперед, поскольку блок команд, который в программах из главы 2 нам приходилось перепрыгивать, процессор ARM позволяет выполнять условно (как в программе на рис. 3.15). Параметры обеих подпрограмм для процессора RAM передаются через регистры.

```
Подпрограмма
INSERTION  CMP     RHEAD,#0                 Проверка того, пуст ли список
           MOVEQ   RHEAD,RNEWREC            Если пуст, он вставляется
           MOVEQ   PC,R14                   в начало новой записи
           LDR     R0,[RHEAD]               Если не пуст, проверяется,
           LDR     R1,[RNEWREC]             должна ли новая запись стать
           CMP     R0,R1                    первой, и, если да,
           STRGT   RHEAD,[ RNEWREC,#4]      вставляем ее
           MOVEQ   RHEAD,RNEWREC
           MOVEQ   PC,R14
           MOV     RCURRENT,RHEAD           Если новая запись не должна
LOOP       LDR     RNEXT,[RCURRENT,#4]      стать первой, ищем,
           CMP     RNEXT,#0                 куда ее следует вставить
           STRGT   RNEWREC,[RCURRENT,#4]    Новая запись становится
                                            последней
           MOVEQ   PC,R14
           LDR     R0,[RNEXT]               Идем дальше?
           CMP     R0,R1
           MOVLT   RCURRENT,RNEXT           Да, затем переходим
                                            на начало цикла
           BLT     LOOP
           STR     RNEXT,[RNEWREC,#4]       Иначе между текущей
           STR     RNEWREC,[RCURRENT,#4     и следующей записями
           MOV     PC,R14                   вставляется новая запись
```

Рис. 3.16. Подпрограмма для процессора ARM, вставляющая новый элемент в связный список

Для ссылки на регистры вместо обычных обозначений R*i* могут использоваться мнемонические обозначения, соответствующие назначению регистров в программе. Для определения мнемонических имен регистров предназначена ассемблерная директива RN. Как и в программах на рис. 2.37 и 2.38, в регистре RHEAD хранится адрес первой записи списка, а в регистре RNEWREC — адрес вставляемой записи. В регистре RIDNUM содержится код удаляемой записи, а в регистрах RCURRENT и RNEXT — адреса связывания, используемые подпрограммами для перемещения по списку и поиска позиций вставляемой и удаляемой записей.

Подпрограмма вставки, приведенная на рис. 3.16, соответствует программе, которую вы видите на рис. 2.37, и имеет такую же структуру. Первые три ее команды вставляют новую запись в начало (оно же конец) пустого списка. Напомним, что первоначально в поле связывания новой записи содержится нуль. Третья команда в этом блоке выполняет операцию возврата из подпрограммы в вызывающую программу. Следующие шесть команд определяют, должна ли новая запись стать новым началом непустого списка. Список упорядочен по возрастанию значений кодов записей. Поэтому, если код записи, содержащийся в первом слове первой

записи списка, больше кода новой записи, то новая запись становится первой записью списка. В этом случае связывание элементов списка производится при помощи условно выполняемых команд STRGT и MOVGT. Если же запись вставляется не в начало списка, оставшаяся часть подпрограммы определяет место ее вставки, включая и вариант вставки ее в конец списка.

```
Подпрограмма
DELETION    LDR     R0,[RHEAD]          Проверка того, является ли
            CMP     R0,RIDNUM           удаляемая запись первой
            LDREQ   RHEAD,[RHEAD,#4]    Если да, удаляем ее
            MOVEQ   PC,R14              и выходим из подпрограммы
            MOV     RCURRENT,RHEAD      В противном случае
                                        продолжаем поиск
LOOP        LDR     RNEXT,[ RCURRENT,#4] Является ли следующая
            LDR     R0,[RBEXT]          запись удаляемой?
            CMP     R0,RIDNUM
            LDREQ   R0,[RNEXT,#4]       Если да, удаляем ее
            STREQ   R0,[RCURRENT,#4]    и выходим из подпрограммы
            MOVEQ   PC,R14
            MOV     RCURRENT,RNEXT      В противном случае
                                        осуществляется
            B       LOOP                переход к началу цикла для
                                        продолжения поиска
```

Рис. 3.17. Подпрограмма для процессора ARM, удаляющая элемент из связного списка

Подпрограмма удаления элемента списка приведена на рис. 3.17. Если удаляемая запись является первой записью списка, первые четыре команды обнаруживают это, удаляют ее и производят выход из подпрограммы. В противном случае оставшаяся часть подпрограммы просматривает список в поисках удаляемой записи, используя регистры RCURRENT и RNEXT. Пара команд LDREQ/STREQ удаляет запись, на которую указывает регистр RNEXT.

Как и в универсальных подпрограммах, приведенных на рис. 2.37 и 2.38, в подпрограмме вставки на рис. 3.16 предполагается, что код новой записи не совпадает ни с одним другим кодом из числа имеющихся в списке записей, а в подпрограмме удаления на рис. 3.17 делается предположение о том, что в списке содержится запись с кодом, указанным в регистре RIDNUM. В упражнениях 3.23 и 3.24 читателю будет предложено изменить эти подпрограммы таким образом, чтобы в случае невыполнения указанных предположений они сообщали об ошибке.

Система команд процессора Motorola 68000

Далее на примере системы команд процессора 68000 будет рассмотрена базовая архитектура семейства процессоров 680X0 компании Motorola. В это семейство входят несколько процессоров с различной производительностью. В основе всех

процессоров 680X0 лежит одна и та же архитектура, но последние процессоры этого семейства имеют дополнительные элементы, повышающие их производительность. Мы выбрали процессор 68000 по той причине, что его легче охарактеризовать, а к тому же он хорошо отражает архитектуру всего семейства. Разумеется, мы не приводим здесь полного описания процессора, так как читатель при желании может найти таковое в документации от производителя, представленной на его web-узле по адресу http://www.motorola.com. Нас интересуют лишь самые важные аспекты архитектуры, позволяющие писать, ассемблировать и выполнять простые программы. Отличительные особенности различных членов семейства 680X0 и некоторые их элементы, связанные с повышением производительности, перечислены в главе 11. Важнейшие возможности рассматриваемого процессора иллюстрируются программами из главы 2, переписанными на языке ассемблера 68000.

3.8. Регистры и адресация

Длина слова процессора 68000 составляет 16 бит, поскольку для соединения с памятью микросхема этого процессора имеет 16 контактов. Однако внутри процессора для работы с данными используются 32-разрядные регистры. Более продвинутыми моделями в этом семействе являются процессоры 68020, 68030 и 68040, выпускаемые в виде микросхем большего размера с 32-разрядными внешними шинами данных. Эти процессоры являются полностью 32-разрядными.

3.8.1. Регистры процессора 68000

Как показано на рис. 3.18, процессор 68000 содержит 8 регистров данных и 8 регистров адреса длиной по 32 бита каждый. Регистры данных используются как регистры общего назначения и как счетчики.

Команды процессора 68000 поддерживают три разные длины операндов. Говорят, что 32-разрядный операнд занимает *длинное слово*, 16-разрядный — слово, а 8-разрядный — байт. Когда в команде используется операнд длиной в один байт или слово, в регистре процессора этот операнд располагается в младших разрядах. Как правило, такие команды не затрагивают оставшиеся старшие разряды регистра, но некоторые из них расширяют знак более короткого операнда на старшие разряды регистра.

В адресных регистрах содержится информация, используемая для определения адресов операндов в памяти. Такого рода информация может быть задана в виде слова или длинного слова. Когда адрес заданного места памяти находится в адресном регистре, то регистр играет роль указателя на это место памяти. И адресные регистры, и регистры данных могут использоваться в качестве индексных регистров. Один из адресных регистров, A7, имеет особое назначение, а именно выполняет функцию указателя стека. Об этом регистре рассказывается в разделе 3.13.

При вычислении адресов используются 32-разрядные значения и 32-разрядные регистры. Однако в процессоре 68000 для доступа к памяти могут быть задействованы только 24 младших бита адреса. Процессоры 68020, 68030 и 68040 имеют по 32 внешние адресные линии и по 32 линии данных.

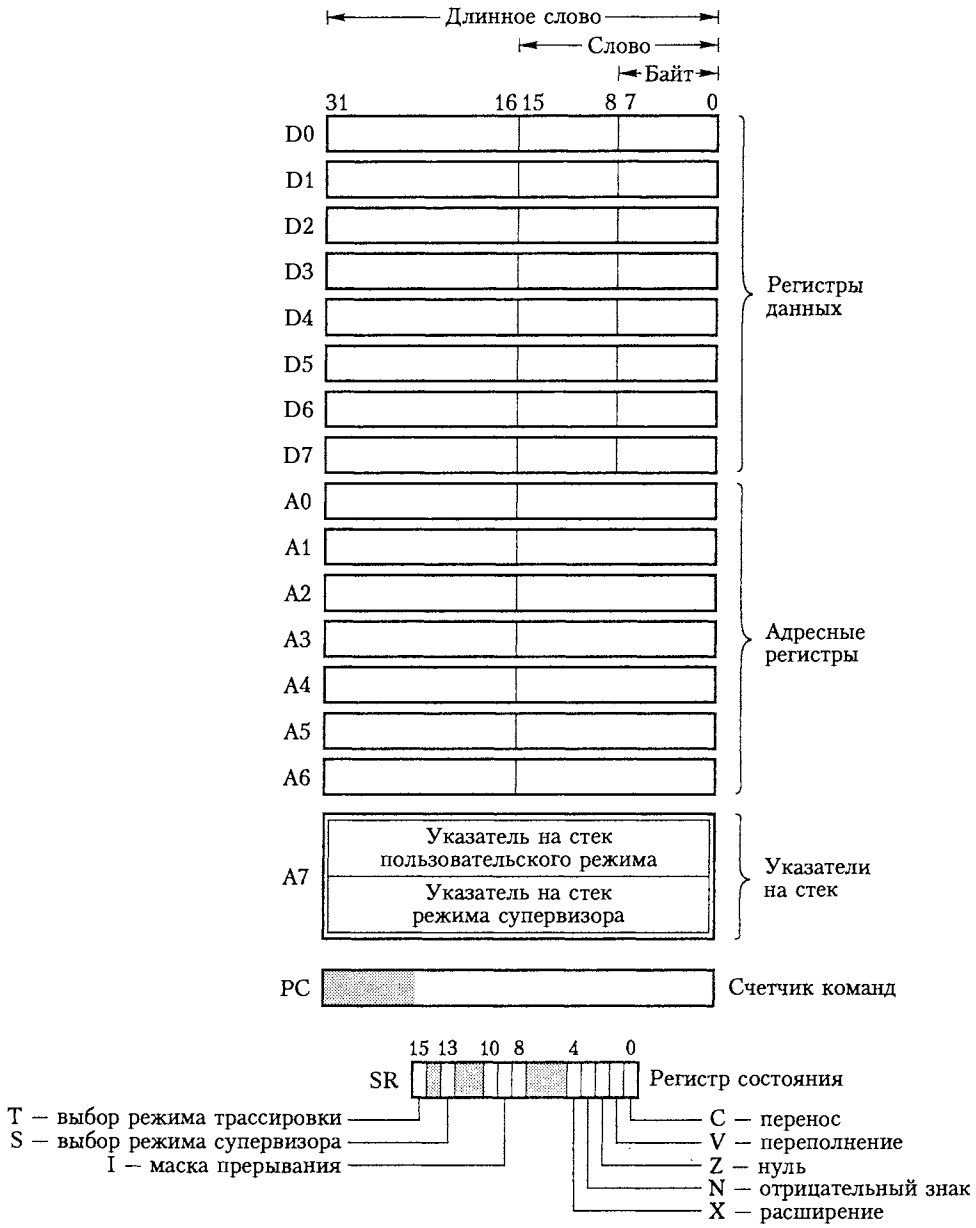


Рис. 3.18. Регистры процессора 68000

Последний из показанных на рис. 3.18 регистров, SR, называется *регистром состояния процессора* (status register). В нем содержатся пять разрядов кодов условий, три разряда прерываний и два разряда выбора режима, описанных соответственно в разделах 3.11.1, 4.2 и 3.13.

3.8.2. Адресация

Память компьютера 68000 организована в виде набора 16-разрядных слов с побайтовой адресацией. Два последовательных слова могут интерпретироваться как одно 32-разрядное длинное слово. Адреса памяти назначаются таким же образом, как на рис. 3.19. Адреса слов должны быть четными. В словах применяется обратный порядок байтов. Это означает, что старший байт слова имеет тот же адрес, что и слово, а адрес его младшего байта на единицу больше.

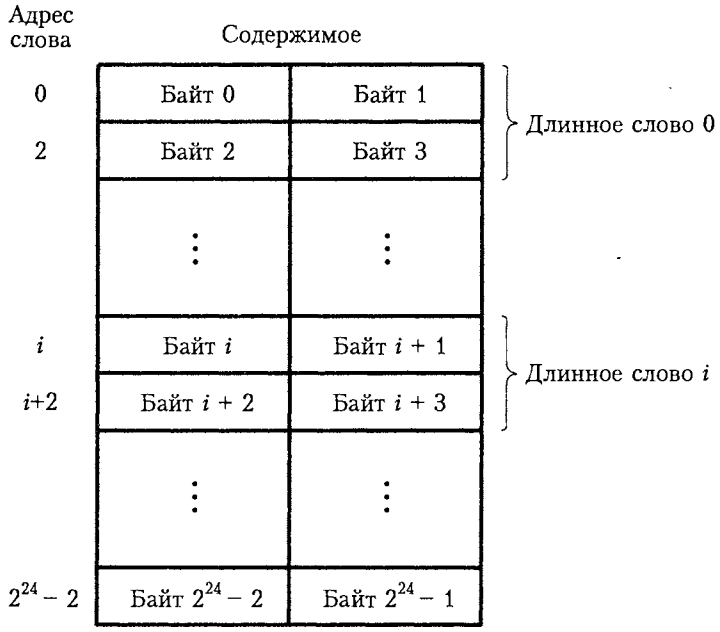


Рис. 3.19. Адресация памяти процессором 68000

Поскольку процессор 68000 генерирует 24-разрядные адреса, его адресное пространство имеет размер 2^{24} (16777216 или 16 М) байт. Его можно представить состоящим из 512 (2^9) страниц по 32 К (2^{15}) байт каждая. Таким образом, шестнадцатеричные адреса от 0 до 7FFF составляют страницу 0, адреса от 8000 до FFFF — страницу 1 и т. д. Последняя страница содержит адреса от FF8000 до FFFFFF.

Процессор 68000 поддерживает несколько режимов адресации, включая режимы, описанные в разделе 2.5. Многие команды этого процессора помещаются в 16-разрядном слове, но некоторым для адресной информации требуются дополнительные слова. Первое слово команды хранит код, определяющий выполняемую командой операцию и содержащий некоторую информацию об адресации. Оставшаяся часть адресной информации хранится в последующих словах. Ниже перечислены все возможные режимы адресации процессора 68000.

- *Непосредственная адресация* (immediate). Операнд содержится прямо в команде. Операнды длиной в байт, слово или длинное слово следуют за словом кода операции. Поддерживается еще один, четвертый, размер операнда, который представляет собой очень маленькое число, включаемое непосредственно в слово кода операции.
- *Абсолютная адресация* (absolute). Абсолютный адрес операнда задается в команде после кода операции. Существует два варианта этого режима — длинный и короткий. В длинном режиме 24-разрядный адрес задается явно, а в коротком режиме заданное в команде 16-разрядное значение используется как 16 младших разрядов адреса. Знаковый бит этого значения распространяется на старшие 8 разрядов адреса. Так как знаковый бит равен 0 или 1, в коротком режиме доступны только две страницы адресного пространства. Это страницы 0 и FF8, каждая размером по 32 Кбайт.
- *Регистровая адресация* (register). Операнд находится в регистре процессора, заданном в команде.
- *Косвенная регистровая адресация* (register indirect). Исполнительный адрес операнда содержится в адресном регистре, заданном в команде.
- *Автоинкрементная адресация* (autoincrement). Исполнительный адрес операнда содержится в заданном в команде адресном регистре, A*n*. После доступа к операнду содержимое регистра A*n* увеличивается на 1, 2 или 4, в зависимости от того, какую длину — байт, слово или длинное слово — имеет операнд
- *Автодекрементная адресация* (autodecrement). Содержимое определенного в команде адресного регистра A*n* уменьшается на 1, 2 или 4, в зависимости от того, какую длину — байт, слово или длинное слово — имеет операнд. Исполнительным адресом операнда является результирующее значение регистра A*n*.
- *Базовая индексная адресация* (basic index). В команде задается 16-разрядное смещение со знаком и адресный регистр A*n*. Исполнительным адресом операнда является сумма значений смещения и регистра A*n*.
- *Полная индексная адресация* (full index). В команде задается 8-разрядное смещение со знаком, адресный регистр A*n* и индексный регистр R*k* (которым может быть либо адресный регистр, либо регистр данных). Исполнительным адресом операнда является сумма значений смещения и регистров A*n* и R*k*. При вычислении адреса используются либо все 32 разряда регистра R*k*, либо 16 младших разрядов с расширенным знаком.
- *Базовая относительная адресация* (basic relative). То же самое, что и базовая индексная адресация, с той лишь разницей, что вместо адресного регистра A*n* применяется счетчик команд.
- *Полная относительная адресация* (full relative). То же самое, что полная индексная адресация, с той лишь разницей, что вместо адресного регистра A*n* применяется счетчик команд.

Все указанные режимы адресации и их синтаксис на языке ассемблера представлены в табл. 3.2.

Таблица 3.2. Режимы адресации процессора 68000

Адресация	Синтаксис языка ассемблера	Формирование адреса
Непосредственная	#Значение	Операнд = Значение
Абсолютная короткая	Значение	EA = СЗначение с расширенным знаком
Абсолютная длинная	Значение	EA = Значение
Регистровая	Rn	EA = R_n, то есть Операнд = [R_n]
Косвенная регистровая	(An)	EA = [A_n]
Автоинкрементная	(An)+	EA = [A_n]; Увеличение значения A_n
Автодекрементная	−(An)	Уменьшение значения Ri; EA = [A_n]
Базовая индексная	СЗначение(An)	EA = СЗначение + [A_n]
Полная индексная	БЗначение(An,Rk.S)	EA = БЗначение + [A_n] + [R_k]
Базовая относительная	СЗначение(PC) или Метка	EA = СЗначение + [PC]
Полная относительная	БЗначение(PC,Rk.S) или Метка(Rk)	EA = БЗначение + [PC] + [R_k]

EA — исполнительный адрес (effective address).

Значение — число, либо заданное явно, либо представленное меткой.

БЗначение — 8-разрядное значение (байт).

СЗначение — 16-разрядное значение (слово).

A_n — адресный регистр.

R_k — адресный регистр или регистр данных.

S — индикатор размера: W для 16-разрядного слова с расширенным знаком или L для 32-разрядного длинного слова.

Обратите внимание на наличие двух версий индексного режима. Базовая индексная адресация соответствует режиму, показанному на рис. 2.13. При полной индексной адресации используется содержимое двух регистров и непосредственно заданное в команде значение смещения. Значение смещения имеет размер 16 бит в базовом режиме и 8 бит в полном.

При полной индексной адресации второй регистр, R*k*, может использоваться двумя способами, с применением либо всех 32 его разрядов, либо только 16 младших. Для выбора одного из них к имени регистра прибавляется суффикс L (длинное слово) или W (слово), например: D1.L, D1.W. Если индикатор размера отсутствует, по умолчанию используется слово. В том случае, если для вычисления 32-разрядного адреса применяется 16-разрядное слово, его знак, как вы понимаете, расширяется до 32 разрядов.

В любом из двух индексных режимов вместо адресного регистра может использоваться счетчик команд. Результирующие режимы адресации называются

относительными, поскольку исполнительный адрес задается как расстояние между операндом и ссылающейся на него командой. Рассмотрим команду

```
ADD  100(PC,A1)D0
```

В машинном коде она состоит из двух слов. В слове кода операции указано, что это команда Add, что регистром назначения является регистр данных D0 и что для доступа к исходному операнду применяется относительная адресация. Во втором слове команды, называемом *словом расширения*, указано, что в качестве индексного регистра используется регистр A1 и что величина смещения (занимает 8 бит) равна 100.

Предположим, что приведенная выше команда хранится по адресу 1000 и что в регистре A1 содержится значение 6, как показано на рис. 3.20. Когда код операции этой команды считывается и декодируется процессором, счетчик команд указывает на слово расширения, то есть содержит значение 1002. Таким образом, исполнительный адрес исходного операнда равен

$$\begin{aligned} EA &= [PC] + [A1] + 100 \\ &= 1002 + 6 + 100 \\ &= 1108 \end{aligned}$$

На рис. 3.20 показано, как использовать описанный режим адресации для доступа к элементам массива. Значение смещения определяет расстояние между первым элементом массива и командой. В индексном регистре задается расстояние между этой точкой и операндом, которым в данном примере является четвертое слово массива.

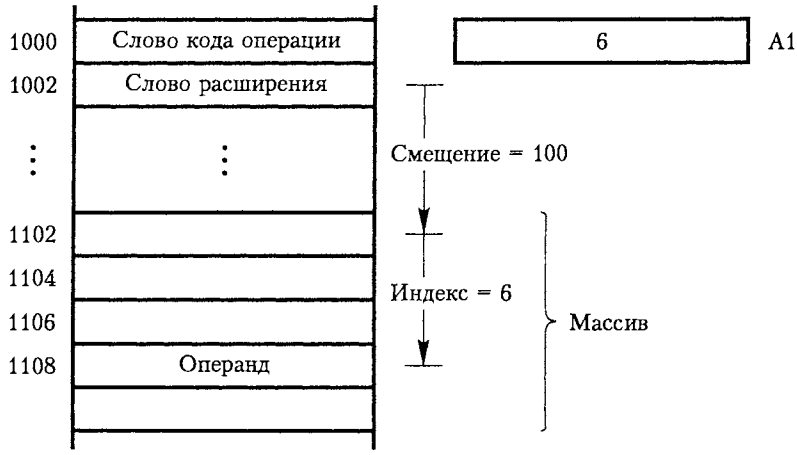


Рис. 3.20. Пример полной относительной адресации процессора 68000 для команды ADD 100(PC,A1)D0

В данном случае относительный режим адресации задан в команде явно. Но большинство ассемблеров позволяет задавать этот режим более простым способом. Прежде всего с помощью специальной директивы ассемблеру необходимо

сообщить, что в данном разделе программы будет использоваться относительная адресация. Далее, после того как имени ARRAY будет присвоено значение 1102, команду на рис. 3.20 можно переписать так:

```
ADD  100(A1),D0
```

В этой команде для исходного операнда задается режим полной относительной адресации, и ассемблер вычисляет величину смещения так, как показано на рисунке. При этом ассемблер не знает и не должен знать, каким будет содержимое регистра A1 во время выполнения команды. Например, эта команда может располагаться внутри цикла, в котором A1 применяется для доступа к элементам массива.

Ограничением полного относительного режима адресации принято считать тот факт, что смещение в нем задается как 8-разрядное число в формате дополнения до двух, то есть его допустимые значения лежат в диапазоне от −128 до +127 байт.

3.9. Команды

Архитектура системы команд процессора 68000 включает расширенный набор команд, в большей части которых могут использоваться операнды трех разных размеров. Упомянутый набор команд приведен в приложении В. Как правило, все эти режимы адресации используются в командах одинаково. Подобные системы команд называются *ортогональными*.

В процессоре 68000 имеются команды с одним и двумя операндами. Команда с двумя операндами имеет следующий синтаксис:

```
OP  src,dst
```

где операция OP выполняется над исходным операндом *src* и операндом назначения *dst*. Результат помещается по адресу назначения. На рис. 3.21 приведен пример команды

```
ADD  #9,D3
```

выполняющей операцию

$$dst \leftarrow [src] + [dst]$$

то есть прибавляющей число 9 к содержимому регистра D3 и сохраняющей результат в регистре D3.

На рис. 3.21, *а* показан общий формат команды ADD. Один из ее операндов должен находиться в регистре данных, D*n*, а другой — либо в регистре, либо в памяти. Возможные комбинации приведены в табл. В.4. Поскольку как минимум один из двух операндов всегда содержится в каком-либо из регистров данных, для его идентификации достаточно 3-битового поля. Второй операнд задается в соответствии с табл. В.1. В нашем примере регистр назначения D3 представлен двоичным значением 011 в разрядах с 9 по 11, а непосредственно заданный исходный операнд представлен значением 111100 в разрядах с 0 по 5.

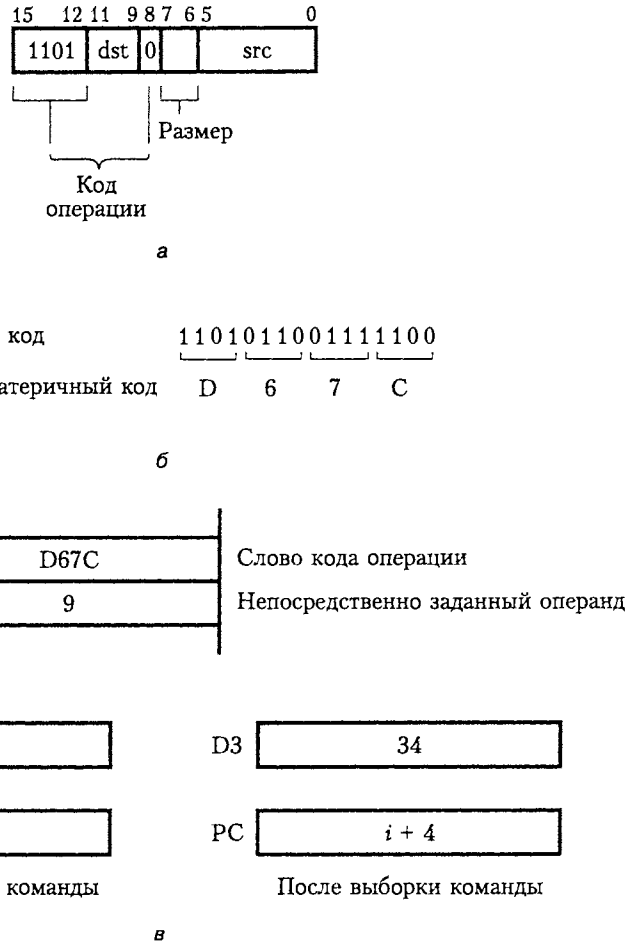


Рис. 3.21. Команда ADD #9,D3 процессора 68000: формат слова кода операции (*а*); кодировка слова кода операции (*б*); последовательность выполнения (*в*)

Размер операнда задается в 2-битовом поле. В нашем примере размер операнда в команде на языке ассемблера не задан явно, поэтому ассемблер по умолчанию считает, что имеет дело с 16-разрядным словом. Согласно табл. В.3, операнды длиной в одно слово обозначаются как 01.

Из сказанного следует, что слово кода операции нашей команды ADD содержит значение 1101011001111100, соответствующее шестнадцатеричному числу D67C (рис. 3.21, *в*). Перед выборкой команды счетчик команд указывает на слово кода операции по адресу *i*. После выборки из памяти очередного слова содержимое PC увеличивается на 2. Поэтому после выполнения команд регистр PC указывает на слово кода операции следующей команды, расположенной по адресу $i + 4$.

Команда вычитания, SUB, имеет тот же формат, что и команда сложения. Она выполняет операцию

$$\text{dst} \leftarrow [\text{src}] - [\text{dst}]$$

Как следует из табл. B.4, команды ADD и SUB обладают достаточной гибкостью, необходимой для определения одного из двух операндов. Однако второй операнд должен находиться в регистре данных. Большинство других команд с двумя операндами имеют те же ограничения. Единственной командой, в которой и исходный и результирующий операнды могут задаваться при помощи большинства поддерживаемых процессором адресных режимов, является команда MOVE, выполняющая операцию

$$dst \leftarrow [src]$$

Рассмотрим простую программу для выполнения задачи C ← [A] + [B], приведенную на рис. 2.8. Эту задачу можно реализовать следующим образом:

```
MOVE   A,D0
ADD    B,D0
MOVE   D0,C
```

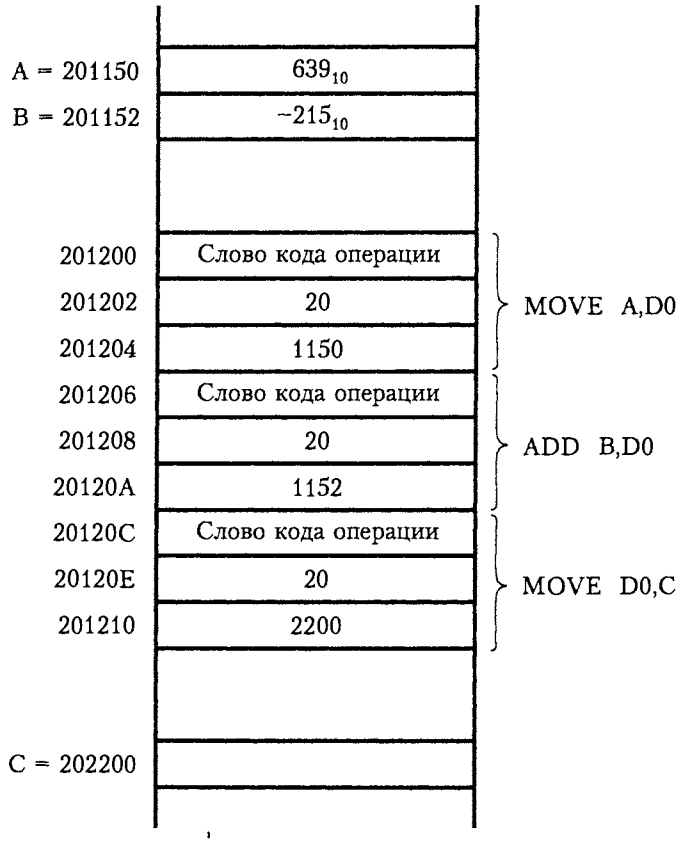


Рис. 3.22. Программа для процессора 68000, выполняющая задачу C ← [A] + [B]

Перечисленные команды могут храниться в памяти компьютера 68000 так, как показано на рис. 3.22, на котором приведены шестнадцатеричные значения адресов и операндов. Операнды имеют длину 16 разрядов, а их адреса задаются в абсолютном режиме. Обратите внимание, что в данном случае требуется длинная версия абсолютного режима, поскольку нужный адрес нельзя представить 16-разрядным значением. Старшие 16 разрядов 32-разрядного адреса помещены, как видите, в младшее слово адреса, а младшие 16 разрядов — в старшее слово, что полностью соответствует соглашению, проиллюстрированному на рис. 3.19.

3.10. Язык ассемблера

Все, что было сказано о языках ассемблера в разделе 2.6, относится и к языку ассемблера процессора 68000. Правда, между ними существуют определенные различия, о которых мы хотели бы поговорить более подробно.

Поскольку команды процессора 68000 могут работать с операндами трех разных размеров, размер операнда должен быть задан в команде ассемблера. Для этого к мнемоническому обозначению команды добавляется индикатор размера: для длинного слова — L, для слова — W, а для байта — B. Если команда Add должна работать, скажем, с длинными словами, она записывается как ADD.L. Но если размер операнда не указан, по умолчанию он принимается равным одному слову. Это означает, что, например, команды ADD.W #20,D1 и ADD #20,D1 идентичны.

Предполагается, что в исходной программе числа задаются в десятичном представлении, если явно не указано иное. Шестнадцатеричное представление обозначается префиксом $, а двоичное — префиксом %. Буквенно-цифровые символы заключаются в одинарные кавычки и заменяются ассемблером их ASCII-кодами. В кавычках может быть задана строка, состоящая из нескольких символов. В частности, вполне допустима строка 'STRING3'.

Все ассемблерные директивы, упоминаемые в разделе 2.6, с незначительными изменениями могут использоваться и в программах для процессора 68000. Начальный адрес блока команд или данных задается при помощи директивы ORG. Директивы EQU связывают имена с числовыми значениями. Константы включаются в объектную программу при помощи директивы DC (Define Constant — определение константы). Для указания размера элементов данных применяется индикатор размера. В одной директиве может быть определено несколько элементов. Так, директивы

```
        ORG    100
PLACE   DC.B   23,$4F,%10110101
```

вызывают загрузку шестнадцатеричных значений 17 (23_{10}), 4F и B5 в память по адресам 100, 101 и 102 соответственно. Метке PLACE присваивается значение 100.

Команда DS (Define Storage — задание области памяти) позволяет зарезервировать блок памяти для данных. Например, команда

```
ARRAY  DS.L  200
```

резервирует 200 длинных слов и ассоциирует с адресом первого из них метку ARRAY.

На рис. 3.23 приведен пример простой ассемблерной программы, соответствующей фрагменту кода на рис. 3.22.

	Метка адреса памяти	Операция	Информация об адресе или данных
Ассемблерные директивы	C	EQU	$202200
		ORG	$201150
	A	DC.W	639
	B	DC.W	−215
		ORG	$201200
Ассемблерные команды,		MOVE	A,D0
для которых генерируются		ADD	B,D0
машинные команды		MOVE	D0,C
Ассемблерная директива		END	

Рис. 3.23. Полная версия программы, приведенной на рис. 3.22, на языке ассемблера 68000

3.11. Управление потоком выполнения программы

Для реализации таких программных структур, как инструкции if и циклы, необходимы команды перехода. В общем случае команда перехода проверяет условие перехода и в зависимости от полученного результата определяет один из двух возможных путей выполнения программы. Проверяемое условие связано с результатом последней реализованной операции.

3.11.1. Флаги кодов условий

У процессора 68000 имеется пять флагов кодов условий, хранящихся в регистре состояния (рис. 3.18), — четыре описанных в разделе 2.4.6 (N, Z, V и C) и один дополнительный, флаг X (extend — расширение). Он устанавливается так же, как флаг C, но не таким большим количеством команд. Это очевидное дублирование удобно для выполнения операций, требующих большой точности, о которых мы поговорим в главе 6.

В приложении B, точнее в табл. B.4, указано, какие флаги соответствуют каждой из команд. Флаги C и X устанавливаются в 1, если в результате операции сложения происходит перенос из самого старшего разряда. Кроме того, они устанавливаются в 1 в том случае, если в результате операции вычитания не производится перенос, что соответствует сигналу отрицательного переноса (заема). Так как длина операндов может быть разной, состояние этих флагов зависит от того, осуществляется ли перенос из разрядов 7, 15 и 31 для операндов длиной в байт, слово

и длинное слово соответственно. Команда MOVE устанавливает флаги N и Z в соответствии со скопированным операндом и очищает флаги C и V. Флаг X не изменяется, если операндом назначения не является сам регистр состояния.

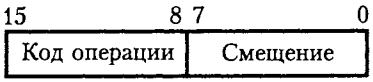
3.11.2. Команды перехода

Команды условного перехода передают управление команде, расположенной по целевому адресу, но лишь в том случае, если выполнено условие перехода. Этот адрес определяется на основе смещения перехода, заданного в поле операнда. Если условие перехода не выполняется, реализация программы продолжается с команды, непосредственно следующей за командой перехода. В системе команд процессора 68000 определены команды перехода с двумя типами смещения. Первый тип — это короткое смещение длиной 8 бит, задаваемое в слове кода операции. Такие команды могут использоваться в тех случаях, когда в момент вычисления адрес перехода расположен на расстоянии от +127 до −128 байт от адреса, указанного в счетчике команд. Напомним, что содержимое счетчика команд увеличивается после выборки из памяти каждого слова, и это означает, что смещение определяет расстояние от слова, следующего за кодом операции команды перехода. Второй тип смещения — это 16-разрядное смещение, задаваемое в слове, следующим за словом кода операции. Им определяется гораздо больший диапазон допустимых значений целевых адресов (±32 Кбайт). В этом случае смещение представляет собой расстояние от слова расширения до целевого адреса перехода.

Пример использования команды перехода с коротким смещением приведен на рис. 3.24. Здесь показано, как реализовать для процессора 68000 программный цикл, представленный на рис. 2.16. Обратите внимание, что в этой программе применяется команда Decrement. Поскольку у процессора 68000 такой команды нет, мы воспользовались командой SUBQ (Subtract Quick — быстрое вычитание), вычитающей непосредственно заданное значение операнда, 1, из содержимого регистра D1. Это значение в виде 3-разрядного операнда включается в слово кода операции команды SUBQ, поэтому для представления данной команды достаточно одного слова.

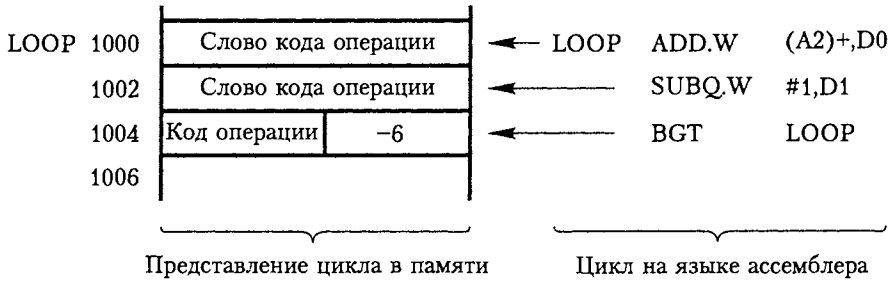
В системе команд процессора 68000 определено целых 16 команд условного перехода, каждая из которых поддерживает 8- и 16-разрядное смещение. Кроме того, имеется команда безусловного перехода BRA, всегда выполняющая переход по заданному адресу. Все эти команды подробно описаны в табл. B.5 и B.6.

На рис. 3.25 представлена программа для процессора 68000, аналогичная приведенной на рис 2.16. В ней используются регистры данных D0 и D1, первый из которых предназначен для накапливания суммы, а второй служит счетчиком, а также адресный регистр A2, указывающий на выбираемые из памяти операнды. Обратите внимание на то, что в данном случае задействован адресный регистр, а не регистр данных, поскольку в автоинкрементном режиме адресации допускается применение только адресных регистров.



Адрес перехода = [обновленный PC] + смещение

а



В момент вычисления целевого адреса перехода [PC] = 1006
Адрес перехода = 1006 − 6 = 1000

б

Рис. 3.24. Команды перехода процессора 68000 с малым смещением: формат команды (*а*); пример ее использования в цикле на рис. 2.16 (*б*)

	MOVE.L	N,D	N содержит количество складываемых элементов *n*, а D1 используется как счетчик, определяющий, сколько раз нужно выполнить цикл
	MOVE.L	#NUM1,A	A2 служит указателем на элементы списка. Он инициализируется значением NUM1, то есть адресом первого элемента списка
	CLR.L	D0	D0 применяется для накопления суммы
LOOP	ADD.W	(A2)+,D0	Прибавление очередного числа к D0
	SUBQ.L	#1,D1	Уменьшение значения счетчика
	BGT	LOOP	Если [D1] ≠ 0, переход на начало цикла
	MOVE.L	D0,SUM	Сохранение суммы в SUM

Рис. 3.25. Программа для процессора 68000, аналогичная представленной на рис. 2.16 программе сложения

Команды декремента и перехода

В дополнение к обычным командам перехода процессор 68000 поддерживает ряд более сложных команд с интегрированным счетчиком, используемых для организации циклов. Вот их формат:

Db*cc* D*n*,LABEL

где суффикс *cc* обозначает условие перехода. Например, команда DBGT с суффиксом GT выполняет переход при выполнении условия «если больше». Полный список условий перехода приведен в табл. В.6. Однако в командах перехода описанного типа условие перехода используется иначе, чем в остальных командах перехода.

- Если условие, определяемое суффиксом *cc*, выполняется, то следующей реализуется команда, указанная непосредственно после команды DB*cc*.
- Если же условие, определяемое суффиксом *cc*, не выполняется, то значение 16 младших разрядов регистра D*n* уменьшается на 1. При условии, что результат равен −1, осуществляется переход к команде по адресу LABEL.

Команды DB*cc* мощнее обычных команд перехода, поскольку решение о переходе зависит не от одного, а от двух условий. Если определять то же действие при помощи обычных команд перехода, потребуется последовательность из трех команд: сначала команда перехода проверит условие *cc*, далее команда декремента уменьшит содержимое регистра счетчика, после чего еще одна команда перехода выполнит переход на основе результата операции декремента. Например, команды

```
Db cc  D3,LOOP
Следующая команда
```

эквивалентны такой последовательности команд:

```
       B cc     NEXT
       SUBQ     #1,D3
       BGE      LOOP
NEXT   Следующая команда
```

Команды DB*cc* удобно рассматривать как средство управления циклом, обеспечивающее выход из цикла по достижении определенного условия. Количество повторений цикла зависит от содержимого регистра счетчика, которым в приведенном выше примере является D3.

В одной из команд DB*cc*, DBF (Decrement and Branch if False — декремент и переход, если ложно), используется условие, которое всегда ложно. Поэтому решение о переходе принимается только на основе значения счетчика. Указанная команда удобна для организации циклов, выполняемых строго заданное количество раз. У нее даже имеется второе имя, а именно DBRA (Decrement and Branch Always — декремент и переход всегда).

Для того чтобы продемонстрировать, насколько удобны команды декремента и перехода, мы переписали их с использованием программы, приведенной на рис. 3.25. Ее новый вариант показан на рис. 3.26. В первом случае (рис. 3.25) регистр D1 инициализировался значением n. Однако поскольку команда DBRA выполняет переход, когда значение счетчика больше или равно нулю, в программе на рис. 3.26 регистр D1 инициализируется значением $n - 1$. Общее количество команд в обеих программах одинаково, но во втором случае программа реализуется быстрее, поскольку цикл в ней короче.

```
        MOVE.L   N,D1          Помещаем значение n − 1
        SUBQ.L   #1,D1         в регистр-счетчик D1
        MOVEA.L  #NUM1,A2
        CLR.L    D0
LOOP    ADD.W    (A2)+,D0
        DBRA     D1,LOOP       Цикл назад до [D1] = −1
        MOVE.L   D0,SUM
```

Рис. 3.26. Программа для процессора 68000, которая выполняет ту же задачу, что и программа, представленная на рис. 3.25

3.12. Операции ввода-вывода

Процессор 68000 требует, чтобы все буферы состояния и данных в интерфейсах устройств ввода-вывода адресовались так, как если бы они располагались в основной памяти. Это означает, что программно-управляемый ввод-вывод в компьютере 68000 должен выполняться по принципу, описанному в разделе 2.7.

Предположим, что разряд b_3 в регистре состояния клавиатуры INSTATUS содержит флаг управления вводом SIN. Тогда для ввода символа с клавиатуры можно выполнить следующие команды:

```
READWAIT  BTST.W   #3,INSTATUS
          BEQ      READWAIT
          MOVE.B   DATAIN,D1
```

Команда проверки бита BTST определяет состояние одного бита операнда назначения и устанавливает значение флага кода условия Z равным дополнению до единицы проверенного бита. Позиция проверяемого бита (в нашем примере это b_3) задается первым операндом.

Предположим, что бит b_3 в регистре состояния дисплея OUTSTATUS содержит флаг управления выводом SOUT. В таком случае для вывода на дисплей символа из регистра D1 можно выполнить такие команды:

```
WRITEWAIT  BTST.W   #3,OUTSTATUS
           BEQ      WRITEWAIT
           MOVE.B   D1,DATAOUT
```

На рис. 3.27 приведена программа для процессора 68000, которая считывает с клавиатуры одну строку символов, сохраняет ее в памяти и выводит на дисплей. Эта программа соответствует программе, приведенной на рис. 2.20. Предполагается, что строка заканчивается нажатием клавиши Enter. Символы записываются в память начиная с адреса LOC.

	MOVEA.L	#LOC,A1	Инициализация регистра счетчика A1, чтобы он содержал адрес, по которому в память должен быть записан первый символ строки
READ	BTST.W	#3, INSTATUS	Ожидание появления символа
	BEQ	READ	в буфере клавиатуры DATAIN
	MOVE.B	DATAIN,(A1)	Пересылка символа из DATAIN в память (в результате флаг SIN устанавливается в 0)
ECHO	BTST.W	#3,OUTSTATUS	Ожидание готовности дисплея
	BEQ	ECHO	
	MOVE.B	(A1),DATAOUT	Пересылка только что прочитанного символа в выходной буферный регистр (в результате SOUT устанавливается в 0)
	CMPI.B	#CR,(A1)+	Проверка того, является ли только что прочитанный символ символом возврата каретки (CR). Если нет, переход к началу цикла и считывание очередного символа
	BNE	READ	Значение указателя увеличивается на 1 с целью записи следующего символа

Рис. 3.27. Программа, которая считывает с клавиатуры строку символов и выводит ее на экран

3.13. Стеки и подпрограммы

О принципах реализации стека было рассказано в разделе 2.8. В качестве указателя на таковой обычно используется один из регистров. Для работы со стеком удобно применять автоинкрементный и автодекрементный режимы адресации. Один из регистров, A7, как уже упоминалось, служит указателем на *стек процессора*. Это стек, используемый во всех автоматически выполняемых процессором операциях, например таких, как связывание подпрограмм.

На рис. 3.18 вы видите два разных 32-разрядных регистра с именем A7. Процессор 68000 поддерживает два режима функционирования, а именно пользовательский режим и режим супервизора. В каждом из них применяется своя версия указателя стека процессора A7. В режиме супервизора процессор может выполнять любые машинные команды. В пользовательском режиме некоторые команды, называемые привилегированными, выполняться не могут. Прикладные программы обычно работают в пользовательском режиме, а системное программное обеспечение — в режиме супервизора. Разряд S в регистре состояния процессора определяет, какой из двух режимов функционирования процессора активен в данный момент, а следовательно, какой из двух регистров A7 сейчас используется.

Для вызова подпрограмм предназначена команда BSR (Branch to Subroutine — перейти к подпрограмме). Реализуется она таким же образом, как и любая другая команда перехода, но кроме перехода по указанному адресу отвечает за сохранение в стеке содержимого счетчика команд. Целевым адресом перехода в этом случае

служит первая команда подпрограммы. Когда выполнение подпрограммы завершается, происходит возврат в вызывающую программу при помощи команды RTS (Return from Subroutine — выйти из подпрограммы), которая выталкивает адрес возврата, находящийся на вершине стека, в счетчик команд. Команды BSR и RTS поддерживают механизм связывания подпрограмм, описанный в разделе 2.9.

На рис. 3.28 показано, как приведенную на рис. 3.26 программу можно представить в виде подпрограммы, для которой параметры передаются через регистры. Адрес списка и количество его элементов подпрограмма получает посредством регистров A2 и D1, а результат выполнения операции сложения возвращает в регистре D0.

```
Вызывающая программа
          MOVEA.L   #NUM1,A2    Запись адреса NUM1 в A2
          MOVE.L    N,D1        Помещение количества элементов n в D1
          BSR       LISTADD     Вызов подпрограммы LISTADD
          MOVE.L    D0,SUM      Сохранение суммы в SUM
          Следующая команда
          ⋮

Подпрограмма
LISTADD   SUBQ.L    #1,D1       Установление значения счетчика в n - 1
          CLR.L     D0
LOOP      ADD.W     (A2)+,D0    Накапливание суммы в D0
          DBRA      D1,LOOP
          RST
```

Рис. 3.28. Программа с рис. 3.26, переписанная в виде подпрограммы для процессора 68000 с передачей параметров через регистры

Представленную на рис. 3.26 программу можно переписать в виде подпрограммы с передачей параметров через стек процессора, на который указывает регистр A7 (рис. 3.29). Команды MOVEM (Move Multiple Registers — копирование нескольких регистров) сохраняют и восстанавливают содержимое регистров A2, D1 и D0. Порядок, в котором это делается, показан на рис. 3.29, *б*. Первая команда MOVEM, для которой принят автодекрементный режим адресации, проталкивает содержимое заданных в ней регистров в стек. Вторая команда MOVEM, в которой используется автоинкрементный режим адресации, выталкивает из стека эти же значения в обратном порядке и сохраняет их в регистрах.

Рассмотрим ситуацию, когда одна из вложенных подпрограмм вызывает другую (см. рис. 2.28). Реализация такой программы для процессора 68000 приведена на рис. 3.30. На рис. 3.31 показаны стековые фреймы подпрограмм SUB1 и SUB2. Как видите, главная программа вызывает подпрограмму SUB1. Перед выполнением команды вызова BSR главная подпрограмма проталкивает в стек параметры этой подпрограммы param2 и param1.

(Предполагается, что вершина стека находится на уровне 1, показанном на рис. 3.29, *б*)

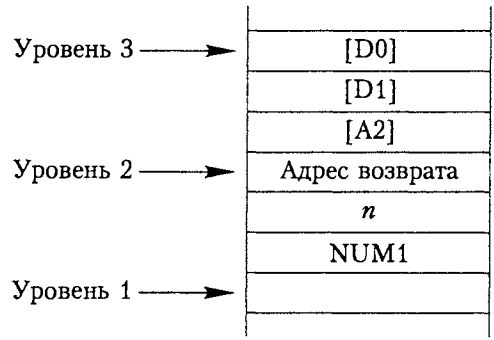
Вызов подпрограммы

```
        MOVE.L    #NUM1,-(A7)     Проталкивание параметров в стек
        MOVE.L    N,-(A7)
        BSR       LISTADD
        MOVE.L    4(A7),SUM       Сохранение результата
        ADDI.L    #8,A7           Восстановление вершины стека
        ⋮
```

Подпрограмма

```
LISTADD MOVEM.L   D0-D1/A2,-(A7)  Сохранение регистров D0, D1 и A2
        MOVE.L    16(A7),D1       Инициализация счетчика значением n
        SUBQ.L    #1,D1           Корректирование значения счетчика
                                  для использования команды DBRA
        MOVEA.L   20(A7),A2       Инициализация указателя на список
        CLR.L     D0              Инициализация суммы значением 0
LOOP    ADD.W     (A2)+,D0        Добавление элемента из списка
        DBRA      D1,LOOP
        MOVE.L    D0,20(A7)       Помещение результата в стек
        MOVEM.L   (A7)+,D0-D1/A2  Восстановление регистров
        RST
```

а



б

Рис. 3.29. Программа с рис. 3.26, переписанная в виде подпрограммы для процессора 68000 с передачей параметров через стек: вызывающая программа и подпрограмма (*а*); содержимое стека в разные моменты времени (*б*)

Адрес памяти		Команды		Комментарии
Вызов программы				
		⋮		
2000		MOVE.L	PARAM2,–(A7)	Помещение параметров в стек
2006		MOVE.L	PARAM1,–(A7)	
2012		BSR	SUB1	
2014		MOVE.L	(A7),RESULT	Сохранение результата
2020		ADDI.L	#8,A7	Сохранение уровня стека
2024		Следующая команда		
		⋮		
Первая подпрограмма				
2100	SUB1	LINK	A6,#0	Установление указателя на фрейм
2104		MOVEM.L	D0–D2/A0,–(A7)	Сохранение регистров
		MOVEA.L	8(A6),A0	Загрузка параметров
		MOVE.L	12(A6),D0	
		⋮		
		MOVE.L	PARAM3,–(A7)	Помещение параметров в стек
2160		BSR	SUB2	Выталкивание результата работы SUB2 в D1
2164		MOVE.L	(A7)+,D1	Помещение результата в стек
		⋮		
		MOVE.L	D2,8(A6)	Восстановление регистров
		MOVEM.L	(A7)+,D0–D2/A0	Восстановление указателя на фрейм
		UNLK	A6	Возврат
		RST		
Вторая подпрограмма				
3000	SUB2	LINK	A6,#0	Установление указателя на фрейм
		MOVEM.L	D0–D1,–(A7)	Сохранение регистров
		MOVE.L	8(A6),D0	Загрузка параметров
		⋮		
		MOVE.L	D1,8(A6)	Помещение результата в стек
		MOVEM.L	(A7)+,D0–D1	Восстановление регистров
		UNLK	A6	Восстановление указателя на фрейм
		RST		Возврат

Рис. 3.30. Вложенные подпрограммы на языке ассемблера процессора 68000

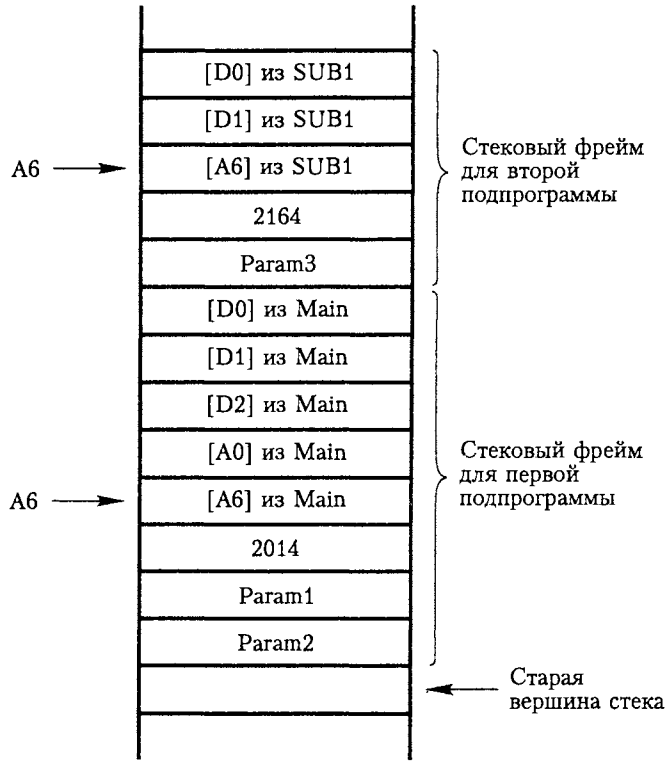


Рис. 3.31. Стековые фреймы для программы, представленной на рис 3.30

Подпрограмма SUB1 начинает свою работу с создания собственного стекового фрейма. Специальная команда

LINK Ai,#смещение

делает регистр A*i* указателем на стек, для чего она выполняет операции, перечисленные ниже.

1. Проталкивает в стек процессора содержимое регистра A*i*.
2. Копирует содержимое указателя стека процессора A7 в регистр A*i*.
3. Прибавляет к содержимому регистра A7 заданное значение смещения.

Если смещение является отрицательной величиной, вершина стека сместится вверх (в направлении уменьшения адресов), в результате чего в стеке образуется пустое пространство, которое может использоваться подпрограммой для хранения локальных переменных. Для доступа к этим переменным удобно применять индексный режим адресации на базе регистра указателя стекового фрейма A*i*. В конце подпрограммы команда ULINK выполняет действия, противоположные действиям команды LINK. Она загружает в регистр A7 содержимое регистра A*i*, опуская вершину стека до ее исходного положения. Затем она выталкивает из стека содержимое регистра A*i* обратно в этот же регистр.

В программе на рис. 3.30 предполагается, что подпрограммы могут выполнять свои задачи, пользуясь только регистрами, и что им не нужно рабочее пространство в стеке. Поэтому каждая подпрограмма начинается с команды

```
LINK  A6,#0
```

определяющей регистр A6 как указатель на стековый фрейм. При этом регистр A7 указывает на то место, где хранится старое содержимое регистра A6. Данная команда выполняет те же действия, что и две первые команды Move в начале подпрограмм на рис. 2.28. В каждом случае за ней следует команда MOVEM, сохраняющая в стеке необходимые подпрограмме регистры.

Оставшаяся часть программы на рис. 3.30 может служить примером реализации программы, представленной на рис. 2.28, с использованием команд процессора 68000. При выполнении программы, приведенной на рис. 3.30, результаты работы подпрограмм помещаются в стек, как показано на рис. 3.31. Главная программа помещает в стек два параметра, а затем команда BSR помещает в него адрес возврата, 2014. Обратите внимание, что команда BSR занимает всего одно слово по адресу 2012, поскольку смещение подпрограммы SUB1 настолько мало, что для его представления достаточно 8 разрядов. Команды LINK и MOVEM в подпрограмме SUB1 сохраняют содержимое указателя стекового фрейма A6 и еще четырех регистров.

Перед вызовом подпрограммы SUB2 подпрограмма SUB1 проталкивает в стек параметр param3. Команда BSR проталкивает в стек адрес возврата, 2164. Эта команда занимает в памяти два слова, так как смещение целевого адреса не помещается в 8 разрядов. Завершив свою работу, каждая подпрограмма восстанавливает содержимое сохраненных регистров и возвращает управление вызывающему коду. После того как управление возвращается главной программе, результат, помещенный в стек подпрограммой SUB1 (на место параметра param2), сохраняется в памяти по адресу RESULT. Затем командой ADDI.L восстанавливается исходное значение указателя стека A7, определяющее старую вершину стека на рис. 3.31.

3.14. Логические команды

В предыдущих разделах были описаны команды, выполняющие перемещение операндов, и арифметические операции, такие как сложение и вычитание. При этом использовались операнды фиксированной длины, а именно 32, 16 и 8 бит. В некоторых приложениях эти же операции можно производить над данными других размеров, в том числе над отдельными битами, а также выполнять логические операции. Для этих целей процессор 68000 применяет несколько специальных команд. В частности, у него имеются команды, выполняющие логические операции И, ИЛИ и Исключающее ИЛИ, а также команды, позволяющие производить несколько видов сдвига и циклического сдвига операндов.

Ниже логические команды этого процессора будут рассмотрены на примерах. Предположим, что в регистре D1 содержится некоторое 32-разрядное двоичное

значение и что мы хотим определить, записано ли в его битах с b_{18} по b_{14} значение 11001. Это можно сделать при помощи следующих команд:

```
AND.L   #$7C000,D1
CMPI.L  #$64000,D1
BEQ     YES
```

Первая команда выполняет логическую операцию И над отдельными битами двух операндов, записывая результат в регистр D1. Шестнадцатеричное число 7C000 состоит из единиц в разрядах от b_{18} до b_{14} и нулей во всех остальных разрядах. Поэтому в результате операции И пять разрядов, от b_{18} до b_{14}, сохранят исходные значения, а остальные разряды будут очищены (установлены в 0). После этого команда Compare сравнит эти пять разрядов с заданным шаблоном.

Программа упаковки цифр

В качестве еще одного примера применения логической команды давайте повторно рассмотрим использование представленной на рис. 2.31 программы упаковки цифр в двоично-десятичном формате. Код этой программы для процессора 68000 приведен на рис. 3.32. Два байта с ASCII-кодами цифр помещаются в регистры D0 и D1. Затем команда LSL выполняет сдвиг байта в регистре D0 на четыре разряда влево, заполняя младшие четыре разряда регистра нулями. В поле операндов данной команды первым задается количество разрядов, на которое нужно сдвинуть второй операнд. Из табл. B.4 следует, что величина сдвига может быть задана в другом регистре данных. Поэтому, если предварительно записать в регистр D2 значение 4, то эту же операцию можно будет выполнить по-другому:

```
LSL.B  D2,D0
```

Команда ANDI устанавливает четыре старших разряда второго байта в 0. И наконец, при помощи команды OR 4-разрядные значения кодов BCD объединяются и сохраняются в памяти по адресу PACKED.

```
MOVEA.L  #LOC,A0     A0 указывает на данные
MOVE.B   (A0)+,D0    Загрузка первого байта в D0
LST.B    #4,D0       Сдвиг байта влево на 4 разряда
MOVE.B   (A0),D1     Загрузка второго байта в D1
ANDI.B   #$F,D1      Очистка старших четырех разрядов нулями
OR.B     D0,D1       Конкатенация цифр
MOVE.B   D1,PACKED   Сохранение результата
```

Рис. 3.32. Использование логических команд процессора IA-32 для упаковки цифр BCD

3.15. Примеры программ

В этом разделе приведены уже известные вам по главе 2 программы для вычисления скалярного произведения, сортировки байтов и операций над связным списком, переписанные для процессора 68000.

3.15.1. Программа для вычисления скалярного произведения двух векторов

На рис. 2.33 была представлена программа, вычисляющая скалярное произведение двух векторов, AVEC и BVEC. Версию этой программы для процессора 68000 вы видите на рис. 3.33. Эти две программы различаются лишь наличием во втором случае команды DBRA, с помощью которой осуществляется управление циклом. Для того чтобы ее можно было использовать, содержимое регистра-счетчика D0 уменьшается на 1, как рассказывалось в разделе 3.11.2.

Обратите внимание, что команда MULS умножает два 16-разрядных числа и генерирует 32-разрядное произведение. Мы предполагаем, что элементы векторов представлены 16-разрядными словами и скалярное произведение также умещается в 16 разрядах. Все адреса интерпретируются как 32-разрядные значения.

```
        MOVEA.L   #AVEC,A1      Адрес первого вектора
        MOVEA.L   #BVEC,A2      Адрес второго вектора
        MOVE      N,D0          Количество элементов
        SUBQ      #1,D0         Изменение значения счетчика для
                                использования команды DBRA
        CLR       D1            D1 будет применяться для накопления суммы
LOOP    MOVE      (A1)+,D2      Считывание элемента вектора A
        MULS      (A2)+,D2      Умножение его на элемент вектора B
        ADD       D2,D1         Прибавление произведения к сумме
        DBRA      D0,LOOP
        MOVE      D1,DOTPROD
```

Рис. 3.33. Программа для процессора 68000, вычисляющая скалярное произведение двух векторов

3.15.2. Программа сортировки байтов

Теперь обратимся к программе сортировки набора байтов (рис. 2.34). Байты содержат ASCII-коды символов, которые программа сортирует по алфавиту. Список помещается в память по адресам от LIST до LIST + n − 1. Информация о количестве элементов в списке, 16-разрядное значение n, хранится по адресу N.

Команда MOVE процессора 68000 позволяет задавать адреса памяти как для исходного, так и для результирующего операнда. Поэтому, когда мы меняем два элемента списка местами, значение LIST(k) копируется прямо в LIST(j). Благодаря этому можно обойтись без временного регистра R4, использовавшегося в программе на рис. 2.34, так что команды программы несколько реорганизуются.

Еще одним отличием программы для процессора 68000 является применение команды DBRA, с помощью которой мы выходим из внутреннего программного цикла, когда индекс k становится равным 0. Обратите внимание, что эту команду нельзя использовать во внешнем цикле, поскольку конечное значение индекса j равно не 0, а 1.

```
for (j = n-1; j>0; j = j-1)
{for k = j-1; k>=0; k = k-1)
   {if (LIST[k] > LIST[j])
      {TEMP = LIST[k];
       LIST[k] = LIST[j];
       LIST[j] = TEMP;
      }
   }
}
```

а

```
        MOVEA.L   #LIST,A1        Указатель на начало списка
        MOVE      N,D1            Инициализация регистра D1, в котором
        SUBQ      #1,D1           будет храниться индекс внешнего цикла j
OTHER   MOVE      D1,D2           Инициализация регистра D2, в котором
        SUBQ      #1,D2           будет храниться индекс внешнего цикла k
        MOVE.B    (A1,D1),D3      Запись в D3 текущего максимального
                                  значения
INNER   CMP.B     D3,(A1,D2)      Если LIST(k) ≤ [D3],
        BLE       NEXT            обмен не выполнять
        MOVE.B    (A1,D2),(A1,D1) LIST(j) и LIST(k) меняются местами,
        MOVE.B    D3,(A1,D2)      в D3 загружается новое
        MOVE.B    (A1,D1),D3      максимальное значение
NEXT    DBRA      D2,INNER        Уменьшение значений счетчиков k и j
        SUBQ      #1,D1           и выполнение перехода к началу цикла,
        BGT       OUTER           если сортировка не завершена
```

б

Рис. 3.34. Программа сортировки байтов для процессора 68000: на языке C (*а*); на языке ассемблера (*б*)

3.15.3. Подпрограммы вставки и удаления элементов связного списка

На рис. 3.35 приведена подпрограмма для процессора 68000, вставляющая запись в связный список. Она идентична программе, представленной на рис. 2.37. Обратите внимание, что для сравнения значений адресов используется версия команды Compare, названная CMPA, а для сравнения значений данных — ее же версия CMP.

Программа для удаления записей из связного списка приведена на рис. 3.36. Она соответствует программе, которую вы видите на рис. 2.38.

Как и в универсальных подпрограммах на рис. 2.37 и 2.38, в программе вставки, приведенной на рис. 3.36, предполагается, что код новой записи не совпадает ни с одним из кодов, имеющихся в списке записей, а в программе удаления, представленной на рис. 2.37, предполагается, что в списке содержится запись, код которой соответствует значению в регистре RIDNUM.

Подпрограмма

```
INSERTION   CMPA.L    #0,A0       A0 соответствует RHEAD
            BGT       HEAD
            MOVEA.L   A1,A0       A1 соответствует RNEWREC
            RST
HEAD        CMP.L     (A0),(A1)   Сравнение кода новой записи с кодом первой
                                  записи списка
            BGT       SEARCH
            MOVE.L    A0,4(A1)    Новая запись становится первой в списке
            MOVEA.L   A1,A0
            RST
SEARCH      MOVEA.L   A0,A2       A2 соответствует RCURRENT
            MOVEA.L   4(A2),A3    A3 соответствует RNEXT
            CMPA.L    #0,A3
            BEQ       TAIL
            CMP.L     (A3),(A1)
            BLT       INSERT
            MOVEA.L   A3,A2       Переход к следующей записи
            BRA       LOOP
INSERT      MOVE.L    A2,4(A1)
TAIL        MOVE.L    A1,4(A2)
            RST
```

Рис. 3.35. Подпрограмма для процессора 68000, вставляющая новый элемент в связный список

Подпрограмма

```
DELETION    CMP.L     (A0),D1     D1 соответствует RIDNUM
            BGT       SEARCH
            MOVEA.L   4(A0),A0    Удаление первой записи
            RST
SEARCH      MOVEA.L   A0,A2       A2 соответствует RCURRENT
LOOP        MOVEA.L   4(A2),A3    A3 соответствует RNEXT
            CMP.L     (A3),D1
            BEQ       DELETE
            MOVEA.L   A3,A2
            BRA       LOOP
DELETE      MOVE.L    4(A3),D2    D2 соответствует RTEMP
            MOVE.L    D2,4(A2)
```

Рис. 3.36. Подпрограмма для процессора 68000, удаляющая элемент из связного списка

Система команд процессоров IA-32 Pentium

Все процессоры корпорации Intel имеют общее название Intel Architecture (IA). Мы с вами рассмотрим лишь процессоры IA, которые работают с 32-разрядными адресами памяти и 32-разрядными данными. Общее название процессоров этого семейства IA-32, а самые последние их представители носят имя Pentium. Первый процессор архитектуры IA-32 — процессор 80386 — увидел свет в 1985 году. После этого были созданы процессоры 80486 (1989), Pentium (1993), Pentium Pro (1995), Pentium II (1997), Pentium III (1999) и Pentium 4 (2000). Каждый новый процессор обладал более высокой производительностью, что достигалось за счет множества архитектурных усовершенствований и новых решений в микроэлектронной технологии. Об эволюции семейства IA-32 рассказывается в главе 11. Последние его члены поддерживают специализированные команды для управления мультимедийной графической информацией и обработки векторных данных. Об этих аспектах системы команд мы вкратце поговорим в данной главе, а немного подробнее — в главе 11. Набор команд процессоров IA-32 очень большой, поэтому придется ограничиться рассмотрением самых основных команд и режимов адресации, а полную информацию об архитектуре системы команд процессоров IA-32 и их языке ассемблера вы найдете на web-узле Intel, который находится по адресу http://www.intel.com.

3.16. Регистры и адресация

В архитектуре процессоров IA-32 память адресуется побайтово при помощи 32-разрядных адресов, а команды работают с операндами размером 8 и 32 разряда. Эти два размера операндов, согласно терминологии Intel, называются байтом и двойным словом. В первых моделях процессоров Intel 16-разрядный операнд назывался словом. Для хранения информации используется прямой порядок байтов, описанный в разделе 2.2.2. Многобайтовые операнды могут начинаться по любым адресам. Выравниваться как-либо в памяти они не должны.

3.16.1. Структура регистров процессоров IA-32

Регистры процессора IA-32 показаны на рис. 3.37. Обычно восемь 32-разрядных регистров с именами от R0 до R7 являются регистрами общего назначения и используются для хранения операндов-данных или адресной информации. Кроме того, имеется восемь регистров с плавающей запятой для хранения операндов-данных, имеющих размер двойного или четверного (64 разряда) слова. Регистры с плавающей запятой содержат поле расширения (на рис. 3.37 не показано), с учетом которого их длина составляет 80 бит. Дополнительные биты используются для увеличения точности при обработке процессором чисел с плавающей запятой. О представлении и обработке чисел с плавающей запятой подробно рассказывается в главе 6. В настоящей главе мы эту тему не обсуждаем.

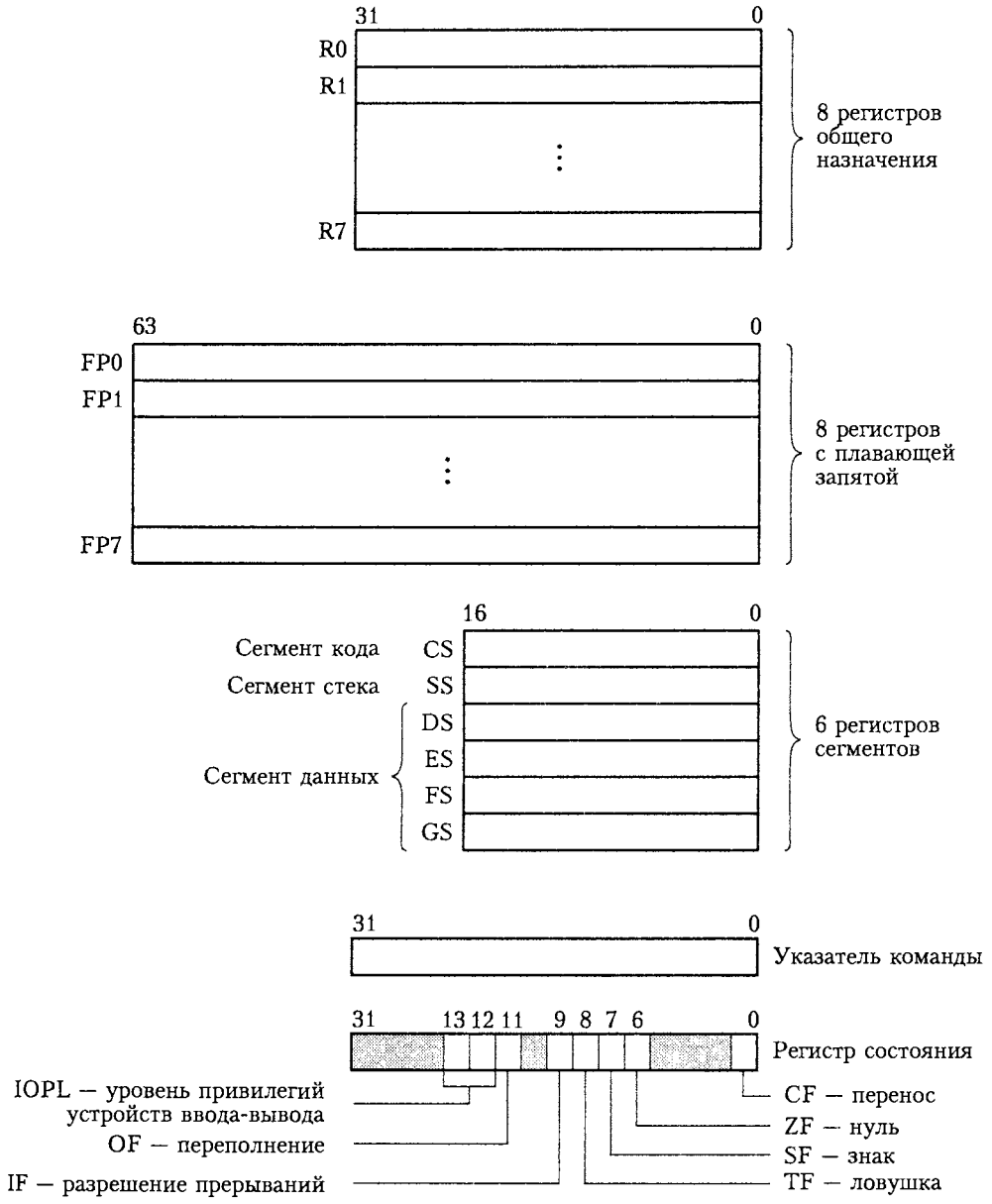


Рис. 3.37. Регистры процессоров IA-32

Архитектура IA-32 основана на модели памяти, в которой различные области памяти, называемые *сегментами*, имеют разное назначение. *Сегмент кода* содержит команды программы, *сегмент стека* — стек процессора, а четыре *сегмента данных* предназначены для хранения операндов-данных. В шести сегментных регистрах хранятся значения селекторов, используемые для идентификации указанных

сегментов в адресном пространстве памяти. О назначении этих регистров мы поговорим в главе 11, где будет обсуждаться семейство IA-32. Пока же такого рода информация нам не нужна. 32-разрядные адреса в архитектуре IA-32 часто применяются для доступа к тем областям памяти, в которых располагается сама программа, стек процессора и области данных.

В нижней части рис. 3.37 показаны еще два регистра — указатель команды, являющийся счетчиком команд и содержащий адрес следующей выполняемой команды программы, и регистр состояния, в котором хранятся флаги кодов условий (CF, ZF, SF, OF). Флаги информируют о результатах арифметических операций (см. раздел 3.19). Биты режима выполнения программы (IOPL, IF, TF) связаны с операциями ввода-вывода и прерываниями, о которых рассказывается в главе 4.

Регистры общего назначения процессоров семейства IA-32 совместимы с регистрами ранних 8- и 16-разрядных процессоров Intel. В этих процессорах на использование разных регистров в программах накладывались некоторые ограничения. О соответствии регистров процессоров IA-32 регистрам более ранних процессоров можно судить по рис. 3.38. Восемь регистров общего назначения разделены на три группы: регистры данных для хранения операндов, регистры-указатели и индексные регистры для хранения адресов и индексов, посредством которых определяется исполнительный адрес операнда в памяти.

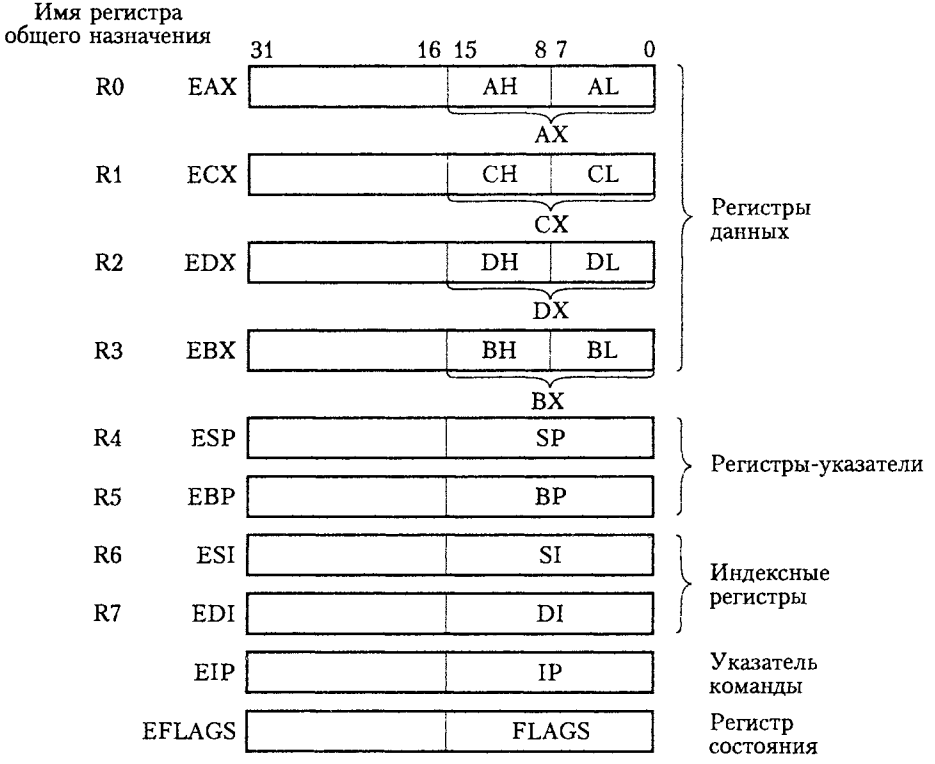


Рис. 3.38. Соответствие регистров IA-32 регистрам более ранних процессоров Intel

В первых, 8-разрядных, процессорах Intel регистры данных назывались A, B, C и D. В более поздних 16-разрядных процессорах их стали называть AX, BX, CX и DX. Старший и младший байты каждого регистра идентифицируются суффиксами H и L. Например, два байта в регистре AX называются AH и AL. В процессорах IA-32 для идентификации соответствующих «расширенных» 32-разрядных регистров используется префикс E: EAX, EBX, ECX и EDX. Этот же префикс употребляется и с другими 32-разрядными регистрами, показанными на рис. 3.38 (они являются расширенными версиями соответствующих 16-разрядных регистров, использовавшихся в более ранних процессорах).

Все перечисленные наименования регистров до сих пор применяются в технической документации Intel и других описаниях процессоров этой корпорации. Старые наименования регистров сохранены потому, что Intel поддерживает обратную совместимость для всех своих процессоров. Это означает, что при корректной установке состояния процессора программы на машинном языке, созданные для ранних 16-разрядных процессоров, будут нормально работать на современных процессорах IA-32 без каких-либо изменений. В программах на языке ассемблера для 16-разрядных процессоров мы будем использовать в именах регистров префикс E, поскольку это мнемоническое обозначение применяется в текущих версиях ассемблера процессоров IA-32. Для однобайтовых операндов, хранящихся в младших разрядах 32-разрядных регистров процессора, используются обозначения AL, BL и т. д. Для команд процессора IA-32 можно динамически установить 32- или 16-разрядный режим, для чего используется байт префикса команды. Об этом мы поговорим в главе 11.

3.16.2. Режимы адресации

В архитектуре IA-32 определен большой и гибкий набор режимов адресации, используемых для доступа к отдельным элементам и областям памяти. Мы приведем полное описание этих режимов и способов их записи на языке ассемблера.

Основные режимы адресации, поддерживаемые большинством процессоров, описаны в разделе 2.5. Речь идет о непосредственной, абсолютной, регистровой и косвенной регистровой адресации. Абсолютная адресация, согласно терминологии Intel, называется прямой; так ее будем называть и мы. Существует несколько режимов адресации, обеспечивающих большую гибкость доступа к операндам, хранящимся в памяти компьютера. Самым гибким из описанных в разделе 2.5 режимов является индексный, обозначаемый как X(R*i*,R*j*). Исполнительный адрес операнда, EA, вычисляется в этом режиме так:

$$EA = [Ri] + [Rj] + X$$

где R*i* и R*j* — это регистры общего назначения, *базовый* и *индексный*, а X — константа, определяющая величину *смещения*. Среди режимов адресации процессоров IA-32 имеется несколько упрощенных разновидностей данного режима.

Полный набор режимов адресации процессоров IA-32 таков.

- *Непосредственная адресация* (immediate). Операнд содержится прямо в команде. Это 8-разрядное или 32-разрядное число, длина которого определяется соответствующим битом в поле кода операции. Для короткого операнда указанный бит равен 0, для длинного — 1.

- *Прямая адресация* (direct). Адрес операнда в памяти определяется заданным в команде 32-разрядным значением.
- *Регистровая адресация* (register). Операнд содержится в одном из восьми регистров общего назначения, заданном в команде.
- *Косвенная регистровая адресация* (register indirect). Адрес операнда в памяти содержится в одном из восьми регистров общего назначения, заданном в команде.
- *Базовая со смещением* (base with displacement). В команде определяются 8- или 32-разрядное смещение со знаком и один из восьми регистров общего назначения, используемый в качестве базового. Исполнительный адрес операнда равен сумме содержимого базового регистра и смещения.
- *Индексная со смещением* (index with displacement). В команде задаются 32-разрядное смещение со знаком, один из восьми регистров общего назначения, который должен использоваться в качестве индексного, и коэффициент масштабирования — 1, 2, 4 или 8. Для получения исполнительного адреса операнда содержимое индексного регистра умножается на коэффициент масштабирования, а к результату прибавляется смещение.
- *Базовая индексная* (base with index). В команде задаются два из восьми регистров общего назначения и коэффициент масштабирования — 1, 2, 4 или 8. Регистры используются как базовый и индексный, а исполнительный адрес операнда вычисляется следующим образом: содержимое индексного регистра умножается на коэффициент масштабирования, а к результату прибавляется содержимое базового регистра.
- *Базовая индексная со смещением* (base with index and displacement). В команде задаются 8- или 32-разрядное смещение со знаком, два из восьми регистров общего назначения и коэффициент масштабирования — 1, 2, 4 или 8. Регистры используются как базовый и индексный, а исполнительный адрес операнда вычисляется следующим образом: содержимое индексного регистра умножается на коэффициент масштабирования, а результат складывается с содержимым базового регистра и смещением.

Синтаксис перечисленных режимов адресации процессоров IA-32 приведен в табл. 3.3. Кроме того, в ней показано, как вычисляется исполнительный адрес операнда для каждого из режимов. Согласно одной из сносок, регистр ESP не может использоваться в качестве индексного, поскольку, как вы увидите далее, он служит указателем на стек процессора. Ниже будет рассмотрено несколько примеров применения режимов адресации процессоров IA-32.

В командах с двумя операндами исходный (src) и результирующий (dst) операнды задаются на языке ассемблера в таком порядке:

КодОперации dst,src

Подобный порядок используется и в архитектуре процессоров ARM, а в процессоре Motorola 68000 применяется противоположный порядок. Так, команда

MOV dst,src

выполняет операцию

dst ← [src]

Таблица 3.3. Режимы адресации процессоров IA-32

Название режима адресации	Синтаксис языка ассемблера	Пример адресации
Непосредственная	Значение	Операнд = Значение
Прямая	Адрес	EA = Адрес
Регистровая	Reg	EA = Reg, то есть Операнд = [Reg]
Косвенная регистровая	[Reg]	EA = [Reg]
Базовая со смещением	[Reg + Disp]	EA = [Reg] + Disp
Индексная со смещением	[Reg * S + Disp]	EA = [Reg] × S + Disp
Базовая и индексная	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] × S
Базовая индексная со смещением	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Значение — 8- или 32-разрядное число со знаком.

Адрес — 32-разрядный адрес.

Reg, Reg1, Reg2 — один из регистров общего назначения (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI), с тем исключением, что регистр ESP не может использоваться в качестве индексного.

Disp — 8- или 32-разрядное число со знаком (смещение), с тем исключением, что смещение в режиме индексной адресации со смещением может быть только 32-разрядным.

S — коэффициент масштабирования, равный 1, 2, 4 или 8.

Для иллюстрации режимов адресации удобно задействовать команду Move. Например, в команде

```
MOV  EAX,25
```

для пересылки десятичного значения 25 в регистр EAX применяется непосредственная адресация. По умолчанию заданное в такой форме число, состоящее из цифр от 0 до 9, интерпретируется как десятичное. Для обозначения двоичных и шестнадцатеричных чисел используются суффиксы B и H. Например, команда

```
MOV  EAX,3FA00H
```

пересылает в регистр EAX шестнадцатеричное число 3FA00.

В команде

```
MOV  EAX,LOCATION
```

используется прямая адресация. Данная команда пересылает в регистр EAX двойное слово из памяти по адресу, определяемому меткой LOCATION. При этом предполагается, что указанная метка определена как метка для адреса памяти в разделе объявлений программы на языке ассемблера. Из раздела 3.18 вы поймете, как это делается. Если метка LOCATION представляет адрес 1000, то данная команда пересылает в EAX двойное слово, расположенное по адресу 100.

Чуть позже мы поговорим о непосредственной и прямой адресации процессоров IA-32 более подробно, поскольку эти режимы часто путают. Возьмем такой случай. Иногда бывает полезно определить символические имена для числовых констант, задаваемых непосредственно в командах. Для назначения константам

символических имен предназначена команда EQU, описанная в разделе 2.6.1. Например, команда

```
NUMBER EQU 25
```

связывает символическое имя NUMBER с десятичным числом 25. После этого команда

```
MOV EAX,NUMBER
```

интерпретируется ассемблером как операция пересылки в регистр EAX непосредственно заданного операнда NUMBER. С другой стороны, если определить NUMBER как адресную метку, этот же операнд будет интерпретироваться как заданный при помощи прямого режима адресации.

Во многих языках ассемблера во избежание такой неоднозначности используется специальный символ, например «#», прибавляемый к числу в качестве префикса для обозначения непосредственной адресации. В языке ассемблера процессоров IA-32 для этой цели могут использоваться квадратные скобки:

```
MOV EAX,[LOCATION]
```

Однако если метка LOCATION определена как адресная, квадратные скобки не нужны.

При необходимости интерпретировать адресную метку как непосредственно заданный операнд можно воспользоваться ассемблерной директивой OFFSET. Так, команда

```
MOV EBX,OFFSET LOCATION
```

помещает значение адресной метки LOCATION, предположим 1000, в регистр EBX с применением непосредственной адресации. После этого EBX может использоваться в косвенном регистровом режиме адресации в команде

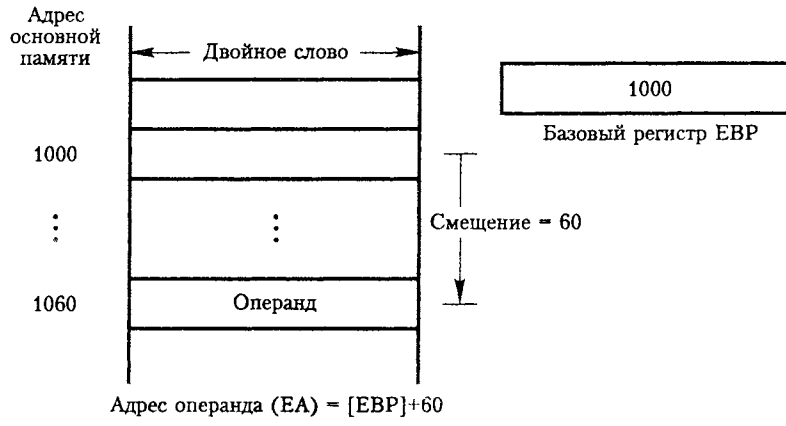
```
MOV EAX,[EBX]
```

пересылающей в регистр EAX содержимое памяти по адресу LOCATION, содержащемуся в регистре EBX. Слово OFFSET (смещение), выбранное для этой директивы языка ассемблера, подчеркивает, что адрес всегда интерпретируется как относительное расстояние от начальной точки сегмента памяти, содержащего операнд команды. Во всех приведенных выше примерах операнд назначения задавался в регистровом режиме адресации.

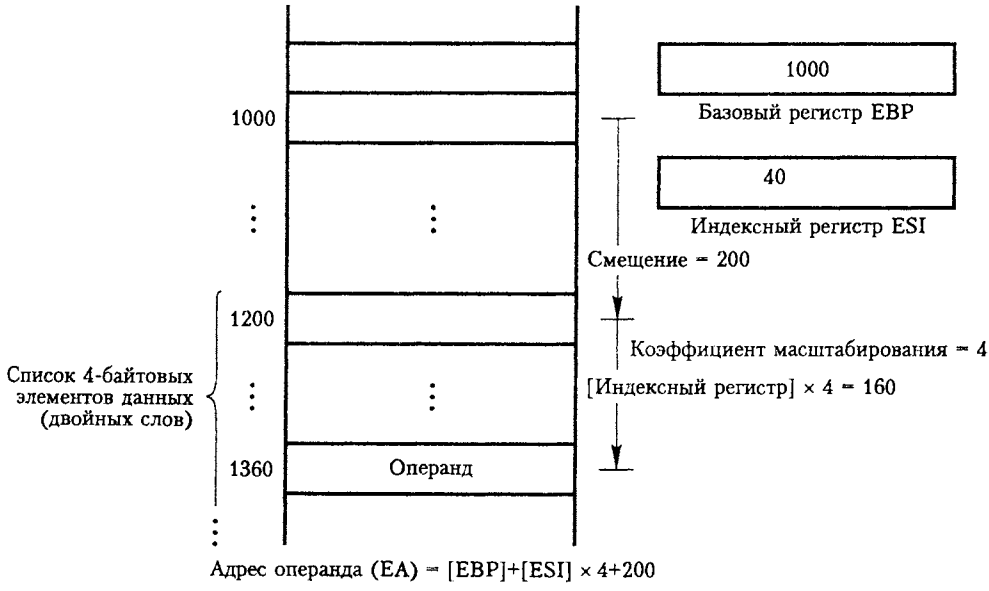
Мы проиллюстрировали использование четырех основных типов адресации: непосредственной, прямой, регистровой и косвенной регистровой. Оставшиеся четыре режима предназначены для более гибкого доступа к операндам, хранящимся в памяти компьютера.

На рис. 3.39, *а* показано, как используется режим базовой адресации со смещением. Базовым регистром здесь служит EBP. Двойное слово, расположенное на расстоянии 60 байт от базового адреса 1000, то есть по адресу 1060, можно переслать в регистр EAX при помощи команды

```
MOV EAX,[EBP+60]
```



а



б

Рис. 3.39. Примеры типов адресации в архитектуре IA-32: базовая со смещением, заданная как [EBP+60] (*а*); базовая индексная со смещением, заданная как [EBP+ESI*4+200] (*б*)

Команды и режимы адресации процессоров IA-32 могут использоваться для работы как с отдельными байтами, так и с двойными словами. Если в базовом регистре EBP содержится, скажем, адрес 1000, то для загрузки в младший байт регистра EAX одного байта, хранящегося по адресу 1010, можно воспользоваться командой

```
MOV  AL,[EBP+10]
```

Поскольку операнд назначения AL представляет собой младший байт регистра EAX, ассемблер выберет код той операции пересылки, которая предназначена для пересылки байтов.

Наиболее гибкой из всех адресаций является базовая индексная со смещением. Пример ее использования приведен на рис. 3.39, *б*, где функции базового и индексного регистров выполняют регистры EBP и ESI. Этот пример показывает, как обратиться к операнду, который является одним из элементов списка двойных слов. Список начинается со смещением 200 относительно базового адреса 1000. С использованием коэффициента масштабирования 4 для доступа к двойным словам по адресам 1200, 1204, 1208, ... можно обращаться при помощи последовательности индексов 0, 1, 2, ... в индексном регистре ESI. В нашем случае в индексном регистре содержится значение 40 и при этом выполняется обращение к двойному слову по адресу 1360 (то есть 1000 + 200 + 4 × 40). Данный операнд загружается в регистр EAX командой

```
MOV  EAX,[EBP+ESI*4+200]
```

Использование коэффициента масштабирования облегчает доступ к последовательным двойным словам списка в программном цикле, поскольку в этом случае на каждом шаге цикла достаточно увеличивать значение индексного регистра на 1. Мы подробно обсудили два способа адресации, и теперь вы сможете понять, как функционируют сходные режимы адресации — индексная со смещением и базовая индексная.

Напоследок хотелось бы высказать одно замечание. Может показаться, что режим базовой адресации со смещением (табл. 3.3) является излишним, поскольку тот же результат можно получить при помощи индексной адресации со смещением с коэффициентом масштабирования 1. Однако команда с использованием первого режима на один байт короче. Кроме того, величина смещения во втором случае может задаваться только 32-разрядным числом.

В следующем разделе рассказывается о том, как описанные режимы адресации кодируются в машинных командах. Более подробная информация по данной теме приведена в приложении Г.

3.17. Команды IA-32

Набор команд архитектуры IA-32 очень обширен. Машинные команды имеют переменную длину и не всегда однотипный формат, о чем будет рассказано в разделе 3.17.1. В большинстве команд IA-32 задаются один или два операнда. Если операндов два, только один их них может храниться в памяти, а другой должен находиться в регистре процессора. Наряду с обычными командами для пересылки данных между памятью и регистрами процессора, выполнения арифметических операций в наборе команд IA-32 имеется множество команд для реализации логических операций, операций сдвига и циклического сдвига, а также для манипулирования стеком процессора. Для обработки нечисловых данных предусмотрены команды, оперирующие строками байтов.

Мы начнем с рассмотрения небольшого набора команд и покажем, как они могут использоваться в сравнительно небольшой программе. Команда

```
ADD  dst,src
```

выполняет операцию

$$dst \leftarrow [dst] + [src]$$

а команда

```
MOV  src dst
```

как вы уже знаете, — операцию

$$dst \leftarrow [src]$$

Предположим, что операнды хранятся в регистрах EAX и EBX. Следующие две команды вычисляют сумму операндов в регистре EAX и сохраняют ее в памяти по адресу SUM:

```
ADD  EAX,EBX
MOV  SUM,EAX
```

Поскольку в памяти может располагаться только один из операндов команды, для реализации операции

$$C \leftarrow [A] + [B]$$

с тремя операндами в памяти потребуется целых три команды:

```
MOV  EAX,A
ADD  EAX,B
MOV  C,EAX
```

Команда вычитания

```
SUB  dst,src
```

выполняет операцию

$$dst \leftarrow [dst] - [src]$$

Для увеличения и уменьшения операнда на 1 предназначены команды автоувеличения (инкремента) и автоуменьшения (декремента), INC и DEC.

Прежде чем мы перейдем к рассмотрению программы, выполняющей сложение чисел, вам нужно познакомиться еще с двумя командами. Первая из них — команда перехода. Если результат последней арифметической операции был больше нуля, команда

```
JG  LOOPSTART
```

осуществляет переход по адресу LOOPSTART. Все команды условного перехода начинаются с буквы J, соответствующей слову Jump (переход), за которой следуют буквы, обозначающие условие. В данном примере буква G обозначает «больше нуля». О других командах условного перехода будет рассказано чуть позже.

Чтобы получить возможность при косвенной регистровой адресации использовать регистр общего назначения, нужно сначала загрузить в такой регистр адрес операнда команды. Это можно сделать двумя способами. Если для нужного адреса в программе явно определена адресная метка, скажем LOCATION, этот адрес можно загрузить в регистр при помощи команды Move и с применением непосредственной адресации, вот так:

```
MOV  EBX,OFFSET LOCATION
```

Данная команда загружает в регистр EBX адрес, представленный меткой LOCATION. В качестве альтернативы можно задействовать команду Load Effective Address (загрузка исполнительного адреса) с мнемоническим обозначением LEA. Команда

```
LEA  EBX,LOCATION
```

выполняет ту же операцию, что и приведенная выше команда. При помощи команды LEA можно загружать в регистры адреса, динамически вычисляемые во время выполнения программы. Предположим, нам нужно загрузить в регистр EBX адрес операнда, для доступа к которому применяется базовая адресация со смещением. Команда

```
LEA  EBX,[EBP+12]
```

загрузит в регистр EBX адрес операнда, расположенного по адресу [EBP]+12. Адрес зависит от содержимого регистра EBX в момент выполнения команды.

Программа для сложения чисел

Используя только что описанные команды, мы можем создать программу для сложения чисел при помощи цикла. Предположим, что в памяти по адресу N содержится набор 32-разрядных чисел, расположенных последовательно начиная с адреса NUM1. На рис. 3.40, *а* приведена программа на языке ассемблера, складывающая эти числа и помещающая результат в память по адресу SUM.

В регистр EBX загружается значение адреса NUM1. Он используется как базовый регистр при базовой индексной адресации в первой команде цикла, расположенной по адресу STARTADD. Индексным регистром здесь является EDI. Перед началом цикла он очищается путем записи в него значения 0. На первой итерации цикла к содержимому регистра EAX, первоначально установленному в 0, прибавляется первое число из списка, хранящееся по адресу [EBX] + NUM1. Затем значение индексного регистра увеличивается на 1. На второй итерации цикла к содержимому регистра EAX прибавляется второе 32-разрядное число, расположенное по адресу NUM1 + 4, поскольку в команде Add задан коэффициент масштабирования 4. На следующих итерациях к сумме прибавляются числа, находящиеся по адресам NUM1 + 8, NUM1 + 12, ... В качестве счетчика используется регистр ECX. Сначала вторая команда программы загружает в него значение, хранящееся в памяти по адресу N. Далее на каждой итерации цикла значение в этом регистре уменьшается на 1. Команда условного перехода JG выполняет переход к началу цикла по адресу STARTADD, если [ECX] > 0. Когда содержимое

регистра ECX становится равным нулю, это означает, что все числа списка сложены. В таком случае переход не осуществляется и следующая команда пересылки сохраняет содержимое регистра EAX в памяти по адресу SUM.

```
            LEA     EBX,NUM1        Инициализация регистров базы (EBX)
            MOV     ECX,N           и счетчика (ECX)
            MOV     EAX,0           Очистка регистров суммы (EAX)
            MOV     EDI,0           и индекса (EDI)
STARTADD:   ADD     EAX,[EBX+EDI*4] Прибавление очередного числа к EAX
            INC     EDI             Увеличение значения индексного
                                    регистра
            DEC     ECX             Уменьшение значения регистра счетчика
            JG      STARTADD        Если [ECX] > 0, переход на начало цикла
            MOV     SUM,EAX         Сохранение суммы в памяти
```

а

```
            LEA     EBX,NUM1        Инициализация регистра базы EBX
            SUB     EBX,4           и вычитание значения 4, чтобы в нем
                                    содержалось NUM1 – 4
            MOV     ECX,N           Инициализация счетчика ECX
            MOV     EAX,0           Очистка регистров суммы (EAX)
STARTADD:   ADD     EAX,[EBX+ECX*4] Прибавление очередного числа к EAX
            LOOP    STARTADD        Уменьшение значения регистра ECX и,
                                    если ECX] > 0, переход на начало цикла
            MOV     SUM,EAX         Сохранение суммы в памяти
```

б

Рис. 3.40. Программа для процессоров IA-32, выполняющая сложение списка чисел: простая программа (*а*); усовершенствованная версия программы (*б*)

Приведенную на рис. 3.40, *а* программу можно переписать более компактно. Обратите внимание на команды

```
DEC  ECX
JG   STARTADD
```

в конце программного цикла. Они очень часто выполняются в конце циклов. Поэтому в набор команд процессора IA-32 включена команда, объединяющая их функции. Команда

```
LOOP  STARTADD
```

сначала уменьшает содержимое регистра ECX, а затем выполняет переход по адресу STARTADD, если значение ECX не равно нулю. Вторая возможность сократить размер программы связана с двумя регистрами, EDI и ECX, которые использовались в качестве счетчиков. Если мы будем сканировать список складываемых

чисел в обратном направлении, то есть от конца к началу, то нам потребуется только один счетчик. Воспользуемся регистром ECX, поскольку он неявно применяется в команде LOOP. Так как [N] = *n*, последовательные значения регистра EDI в первой программе равны 0, 1, 2, ..., *n*, и программа считывает числа по адресам NUM1, NUM1 + 4, NUM1 + 8, ..., NUM1 + 4(*n* − 1). Новая программа считывает числа по адресам (NUM1 − 4) + 4*n*, (NUM1 − 4) + 4(*n* − 1), ..., (NUM1 − 4) + 4(1), а регистр ECX в ней принимает значения *n*, *n* − 1, ..., 1 (рис. 3.40, *б*). Поэтому значение в базовом регистре EBX, в первой программе равное NUM1, нужно изменить на NUM1 − 4, что позволит учесть различие между значениями регистров EDI и ECX. На последней итерации цикла новой программы перед выполнением команды LOOP регистр ECX содержит значение 1, и из памяти по адресу NUM1 считывается последнее из складываемых чисел.

Обработка списков и массивов всегда требует внимательности, особенно при выборе условия перехода и метода индексации — с нуля или единицы. Неверный выбор может привести к ошибкам в программе. В языках высокого уровня эта задача упрощается, поскольку к элементам списка можно явно обращаться как к LIST(0), LIST(1), ..., LIST(*n*−1), а начало и конец цикла можно явно связать со значениями индекса при помощи таких выражений, как

```
FOR I FROM 0 UPTO (n-1)
```

или

```
FOR I FROM (n-1) DOWNTO 0
```

Это краткое описание некоторых из числа наиболее часто используемых команд IA-32 и примера программного цикла можно считать введением в систему команд и язык ассемблера IA-32. В следующем разделе рассматривается машинное представление команд этого семейства процессоров.

3.17.1. Формат машинных команд

Общий формат машинных команд процессоров IA-32 показан на рис. 3.41. Команды имеют переменную длину — от 1 до 12 байт и могут включать до четырех полей. Команды длиной в 1 байт содержат только обязательное поле кода операции. Как правило, длина этого поля составляет 1 байт, иногда — 2 байта. Информация о режиме адресации содержится в одном или двух байтах, следующих за полем кода операции. В тех командах, где для формирования исполнительного адреса операнда используется только один регистр, поле режима адресации имеет длину 1 байт. Второй байт нужен для кодирования двух последних режимов адресации (см. табл. 3.3). В этих режимах для формирования исполнительного адреса операнда нужно иметь 2 байта.

Если для вычисления исполнительного адреса операнда требуется указать смещение, его значение записывается в один или два байта в поле, следующем за полем режима адресации. Если один из операндов задается непосредственно, он помещается в последнее поле команды и занимает 1 или 4 байта.

Для ряда простых команд, подобных тем, о которых рассказывалось в предыдущем разделе, код используемого регистра задается прямо в байте кода операции.

Однако для большинства команд и режимов адресации регистры задаются в поле режима адресации.

Во многих системах команд в тех случаях, когда таковые имеют переменную длину, в начале двоичного представления команды должна быть задана ее длина. Связано это с тем, что последовательные команды располагаются в памяти друг за другом, без указания границ между ними.

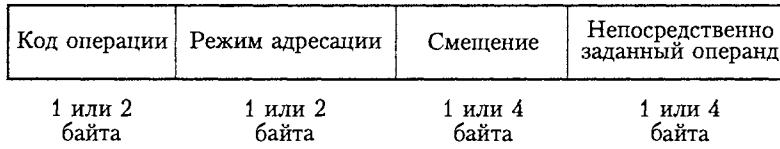


Рис. 3.41. Формат команды IA-32

Однобайтовые команды

Команды INC и DEC, предназначенные для уменьшения и увеличения значения регистра, имеют длину 1 байт. Например, в командах

```
INC EDI
```

и

```
DEC ECX
```

регистры общего назначения EDI и ECX задаются 3-битовыми кодами в байте кода операции.

Кодировка непосредственной адресации

Режим непосредственной адресации задается в поле кода операции. Например, команда

```
MOV EAX,820
```

имеет длину 5 байт. В однобайтовом поле кода операции определяется операция пересылки, указывается, что операнд задан непосредственно и имеет длину 32 разряда, а также задается имя регистра назначения. За кодом операции следует 4-байтовое значение 820. Если непосредственно заданный операнд имеет длину 8 бит, то для команды

```
MOV DL,5
```

достаточно 2 байт.

Режимы адресации и поля смещения

Если команда имеет два операнда, то один из них должен содержаться в регистре, а другой — либо в регистре, либо в памяти. Для этого правила имеются два исключения, когда оба операнда могут находиться в памяти. Одно из них касается команд, в которых первый операнд задается в режиме непосредственной адресации, а второй — в режиме прямой адресации. Другое исключение относится к операциям проталкивания и выталкивания значений из стека процессора. Стек

располагается в памяти в сегменте стека, и в него можно поместить значение из памяти, а можно вытолкнуть из него значение в память. Эти операции подробно описаны в разделе 3.22.

Когда оба операнда находятся в регистрах, для поля режима адресации достаточно одного байта. Например, команда

```
ADD  EAX,EDX
```

занимает 2 байта. В первом из них содержится код операции, а во втором — коды двух регистров.

Рассмотрим несколько примеров кодирования команд, один операнд которых располагается в памяти, а другой в регистре. Команда

```
MOV  ECX,N
```

в программах на рис. 3.40 кодируется 6 байтами: один для кода операции, один для поля адресного режима, в котором задается режим прямой адресации и регистр ECX, и четыре байта для адреса памяти N.

Для кодирования команды

```
ADD  EAX,[EBX+EDI*4]
```

в тех же программах требуется 2-байтовое поле режима адресации, поскольку для вычисления исполнительного адреса исходного операнда используются два регистра. Во втором из этих байтов задается коэффициент масштабирования 4. Таким образом, для всех команд требуется три байта, с учетом байта для кода операции.

Теперь рассмотрим команду

```
MOV  DWORD PTR[EBP+ESI*4+DISP],10
```

Директива ассемблера DWORD PTR указывает, что непосредственно заданный операнд 10 имеет длину 32 бита. В других языках ассемблера размер операнда часто определяется мнемоническим обозначением команды. Например, в языке процессора Motorola 68000, описанного выше в этой главе, команда MOVE.B определяет 1-байтовый операнд, а команда MOVE.L — 4-байтовый. Если в команде задано 32-разрядное значение смещения DISP, для ее кодирования требуется 11 байт: один байт для поля кода операции, два — для поля режима адресации и по четыре байта для полей смещения и непосредственно заданного операнда. В табл. 3.3 указано, что смещение может иметь дину 8 или 32 бита. Его размер задается в первом из двух байтов поля режима адресации.

При кодировании команд с двумя операндами спецификации регистровых операндов и операндов в памяти располагаются в строго определенном порядке, и регистровый операнд всегда задается первым. Для различения команды

```
MOV  EAX,LOCATION
```

которая загружает в регистр EAX содержимое памяти по адресу LOCATION, и команды

```
MOV  LOCATION,EAX
```

загружающей в память по адресу LOCATION содержимое регистра EAX, в поле кода операции содержится бит, который называется *битом направления*. Он указывает, какой из операндов является исходным.

Правила кодирования полей кода операции и адресного режима в архитектуре IA-32 довольно сложны, не всегда унифицированы и имеют множество исключений. И хотя это несколько затрудняет для компилятора использование всех возможностей системы команд и режимов адресации, архитектура IA-32, без сомнения, обладает большой гибкостью.

В приложении Г приведен список команд IA-32 с их краткими описаниями и руководство по вводу и выполнению программ на языке ассемблера на персональном компьютере.

3.18. Язык ассемблера IA-32

Представленные на рис. 3.40 программы демонстрируют принципы использования базовых элементов языка ассемблера IA-32, связанные с определением кодов операции, адресацией и применением адресных меток. Как вы уже знаете из раздела 2.6.1, для определения областей данных программы и установления соответствия между символическими именами и реальными физическими значениями адреса существуют специальные директивы ассемблера.

Полный текст ассемблерной программы, которую вы видите на рис. 3.40, *б*, приведен на рис. 3.42. Ассемблерные директивы .data и .code определяют начало областей данных и кода (то есть команд) программы. Директива DD выделяет в области данных 4-байтовое двойное слово. NUM1 — это метка, назначенная адресу первого из двойных слов, которые инициализированы значениями 17, 3, −51, 242 и −113. Адресам следующих двух двойных слов, инициализированных значениями 5 и 0, назначены метки N и SUM.

Директивы ассемблера	.data		
	NUM1	DD	17,3,−51,242,−113
	N	DD	5
	SUM	DD	0
	Код		
Команды, транслируемые в машинные команды	MAIN:	LEA	EBX,NUM1
		SUB	EBX,4
		MOV	ECX,N
		MOV	EAX,0
	STARTADD:	ADD	EAX,[EBX+ECX*4]
		LOOP	STARTADD
		MOV	SUM,EAX
Директивы ассемблера		END	MAIN

Рис. 3.42. Полный текст программы на языке ассемблера, приведенной на рис. 3.40, *б*

Три символических имени, объявленных в разделе данных, используются в командах в разделе кода. В частности, метка MAIN предназначена для определения точки начала выполнения программы, а также задается в ассемблерной директиве END, которой заканчивается текстовый файл программы. В ассемблере IA-32 имеются и другие директивы. Одна из них, EQU, как вы помните, описана в разделе 2.6.1.

3.19. Управление потоком выполнения программы

Существует два способа отклонения программы от «прямолинейного курса». Первый из них (о нем рассказывается в разделе 3.22) — это вызов подпрограммы и возврат из нее. Кроме того, в программе могут выполняться переходы к заданным командам — как условные, так и безусловные. О них мы сейчас и поговорим.

3.19.1. Условные переходы и флаги кодов условий

Представленная на рис. 3.40, *а* команда

```
JG STARTADD
```

относится к числу команд условного перехода. Условие «больше нуля» задается в ней суффиксом кода операции G. Это условие касается и результата последней выполненной команды обработки данных, которой в нашем примере была команда

```
DEC ECX
```

Признаки результатов, которые генерируются командами типа Decrement и Add, выполняющими арифметические операции и операции сравнения, записываются в четыре флага кодов условий в регистре состояния процессора, показанном на рис. 3.37. В зависимости от результата операции эти флаги, называемые SF (sign — знак), ZF (zero — нуль), OF (overflow — переполнение) и CF (carry — перенос), устанавливаются в 1 или очищаются нулем, как рассказывалось в разделе 2.4.6 (там они назывались N, Z, V и C). Но существует одно исключение. В операции вычитания бит CF устанавливается в 1, если перенос не выполняется, что соответствует сигналу обратного переноса. Состояние этих флагов можно проверить в последующих командах условного перехода, с тем чтобы решить, следует ли осуществлять переход. В нашем примере при выполнении условия [ECX] > 0 управление передается команде, записанной по целевому адресу STARTADD.

В команде условного перехода задается не абсолютное значение целевого адреса перехода, а число со знаком, которое прибавляется к содержимому регистра указателя команды, то есть целевой адрес задается относительно адреса в указателе команды. Значение указателя команды увеличивается сразу после выборки очередной команды, поэтому он всегда определяет следующую выполняемую команду программы. Когда к указателю прибавляется относительный адрес перехода,

он начинает указывать на команду, следующую за командой перехода. Предположим, что адрес STARTADD в нашем примере равен 1000. Для кодирования команд ADD, INC, DEC и JG из программы на рис. 3.40, *а* требуется 7 байт. Обновленное содержимое регистра указателя команды EIP будет равно 1007, то есть адресу последней в программе команды MOV. Таким образом, относительное расстояние до целевой команды перехода составляет −7; именно данное значение и содержится в команде условного перехода. Это маленькое отрицательное число можно представить одним байтом. Поэтому, с учетом байта кода операции, для записи команды условного перехода достаточно 2 байт. Такой размер имеют команды перехода, в которых относительный адрес перехода лежит в диапазоне от −128 до +127. Если же расстояние до целевой команды перехода больше, то используется 4-байтовое смещение.

В этом примере проверяется значение в регистре ECX — нас интересует, больше ли оно нуля. Другие свойства результата можно проверить при помощи иных команд условного перехода. Например, если результат равен нулю, переход выполняется командой JZ (или JE), а если знак результата отрицателен (знаковый бит равен 1) — командой JS.

Команды сравнения

Условный переход в программах часто осуществляется в соответствии с результатом сравнения двух чисел. Команда Compare

```
CMP  dst,src
```

выполняет операцию

```
[dst] − [src]
```

и на основе полученного результата устанавливает флаги кодов условий. При этом ни один из операндов не изменяется и первый операнд всегда сравнивается со вторым. Например, переход по условию «больше» выполняется в том случае, если операнд назначения dst больше исходного операнда src.

3.19.2. Безусловный переход

Команда безусловного перехода JMP всегда вызывает переход к команде по целевому адресу. В ней может быть задано короткое (1 байт) или длинное (4 байта) относительное смещение со знаком. Кроме того, как и в командах условного перехода, могут использоваться другие режимы адресации. Такая гибкость определения целевого адреса перехода может быть очень полезной. В каждой точке программы выполняется только одно из альтернативных вычислений. Предположим, что в специальной таблице в памяти начиная с адреса JUMPTABLE хранятся 4-байтовые адреса первой команды каждой из подпрограмм, представляющих возможные ветви программы. Если последовательно пронумеровать эти ветви как 0, 1, 2... и загрузить индекс выполняемой подпрограммы в регистр ESI, то переход к выбранной ветви может быть выполнен посредством такой команды с использованием индексной адресации со смещением:

```
JMP  [JUMPTABLE + ESI*4]
```

3.20. Логические команды, команды сдвига и циклического сдвига

3.20.1. Логические операции

В архитектуре IA-32 имеются команды, выполняющие логические операции И, ИЛИ и Исключающее ИЛИ. Это поразрядные операции с двумя операндами и записью результата по адресу назначения. Предположим, что в регистре EAX содержится шестнадцатеричное значение 0000FFFF, а в регистре EBX — значение 02FA62CA. Команда

```
AND  EAX,EBX
```

очистит левую половину регистра EBX, заполнив ее нулями, а правую его часть оставит без изменений. В результате в EBX окажется значение 000062CA.

Кроме того, в архитектуре IA-32 имеется команда NOT, генерирующая логическое дополнение всех битов операнда, то есть заменяющая все единицы нулями, а все нули единицами.

3.20.2. Операции сдвига и циклического сдвига

При помощи операции логического или арифметического сдвига операнд может быть смещен влево или вправо на заданное количество разрядов. Формат команды сдвига таков:

```
КодОперации  dst,count
```

где сдвигаемый операнд *dst* задается при помощи одного из стандартных адресных режимов, а количество разрядов сдвига *count* представляется 8-разрядным значением, либо задаваемым непосредственно в команде, либо содержащимся в 8-разрядном регистре CL. Существует четыре команды сдвига:

- SHL — логический сдвиг влево;
- SHR — логический сдвиг вправо;
- SAL — арифметический сдвиг влево (то же, что SHL);
- SAR — арифметический сдвиг вправо.

Операции сдвига описаны в разделе 2.10 и проиллюстрированы на рис. 2.30.

Существует также четыре команды циклического сдвига. Это команды ROL и ROR, выполняющие сдвиг влево и вправо без установки флага переноса CF, и команды RCL и RCR, выполняющие сдвиг с установкой флага CF. Все четыре операции графически показаны на рис. 2.32.

Программа упаковки цифр

В качестве простого примера использования указанных команд давайте рассмотрим программу упаковки цифр, представленную на рис. 2.31. Код этой программы для процессоров IA-32 вы видите на рис. 3.43. Два байта ASCII загружаются в регистры AL и BL. Команда SHL сдвигает байт в регистре AL на четыре позиции влево, заполняя четыре освободившихся младших бита нулями. Посредством

второго операнда этой команды задается количество разрядов, на которое должен быть сдвинут первый операнд. Команда AND очищает четыре старших бита второго байта, записывая в них нули. После этого 4-разрядные значения, представляющие BCD-коды чисел, объединяются командой OR в регистре AL, а затем сохраняются в памяти по адресу PACKED.

```
LEA   EBR,LOC     EBP указывает на первый байт
MOV   Al,[EBR]    Загрузка первого байта в AL
SHL   AL,4        Сдвиг на 4 позиции влево
MOV   BL,[EBR+1]  Загрузка второго байта в BL
AND   BL,0FH      Очистка 4 старших разрядов нулями
OR    AL,BL       Конкатенация цифр BCD
MOV   PACKED,AL   Сохранение результата
```

Рис. 3.43. Программа для процессоров IA-32, упаковывающая две цифры BCD в один байт

3.21. Операции ввода-вывода

3.21.1. Ввод-вывод с отображением в память

В современных компьютерных системах буферные регистры ввода-вывода обычно адресуются путем отображения в основную память, как рассказывалось в разделе 2.7. Для пересылки устройствам ввода-вывода управляющей информации, а также для обмена с ними данными и сведениями о состоянии в архитектуре IA-32 может использоваться обыкновенная команда пересылки. Для примера предположим, что флаги синхронизации клавиатуры и дисплея, SIN и SOUT (рис. 2.19), хранятся в третьем разряде регистров состояния этих устройств, то есть в INSTATUS и OUTSTATUS соответственно. При использовании программно-управляемого ввода-вывода чтение байта из буфера клавиатуры DATAIN в регистр AL можно выполнить с помощью такого цикла ожидания:

```
READWAIT  BT    INSTATUS,3
          JNC   READWAIT
          MOV   AL,DATAIN
```

Здесь BT — команда проверки бита. Она загружает значение заданного бита исходного операнда во флаг переноса CF. После этого команда JNC (условный переход при отсутствии переноса) выполняет переход по метке READWAIT, если CF = 0.

Процедура вывода байта из регистра AL на дисплей очень похожа:

```
WRITEWAIT  BT    INSTATUS,3
           JNC   WRITEWAIT
           MOV   DATAOUT,AL
```

На рис. 3.44 приведена программа для процессоров IA-32, считывающая с клавиатуры строку символов, загружающая ее в память по адресу LOC и отображающая ее на экране (рис. 2.20). Регистр EBP указывает на область памяти, в которую должна быть записана строка символов.

```
        LEA     EBP,LOC         EBP указывает на область памяти
READ:   BT      INSTATUS,3      Ожидание ввода символа
        JNC     READ            в DATAIN
        MOV     AL,DATAIN       Пересылка символа в AL
        MOV     [EBP],AL        Сохранение символа в памяти и
        INC     EBP             увеличение значения указателя
ECHO:   BT      OUTSTATUS       Ожидание готовности
        JNC     ECHO            дисплея
        MOV     DATAOUT,3       Отправка символа на дисплей
        CMP     AL,CR           Если это не символ возврата каретки,
        JNE     READ            считывается следующий символ
```

Рис. 3.44. Программа для процессоров IA-32, считывающая с клавиатуры строку символов и отображающая ее на экране

3.21.2. Изолированный ввод-вывод

В наборе команд IA-32 существуют две команды, предназначенные исключительно для ввода и вывода, — IN и OUT. В них задаются адреса памяти, расположенной вне адресного пространства, которое используется другими командами. В отличие от ввода-вывода с отображением в память, когда память устройств ввода-вывода находится в том же адресном пространстве, что и основная память компьютера, описанные операции называются *изолированным вводом-выводом*. Для указания конкретного адресного пространства, используемого в текущей команде, предназначена отдельная выходная управляющая линия.

Для адресуемого побайтово адресного пространства ввода-вывода применяются 16-разрядные адреса. Первые 256 адресов можно задавать непосредственно в 8-разрядном поле команд IN и OUT. Вот какой формат имеет команда ввода, в которой используется этот режим:

```
IN  REG,DEVADDR
```

Здесь регистром назначения REG может служить регистр AL или EAX, что зависит от длины пересылаемого операнда — 8 или 32 разряда. В последнем поле команды содержится 8-разрядный адрес устройства DEVADDR. Соответствующая команда вывода такова:

```
OUT  DEVADDR,REG
```

Поскольку память устройств ввода-вывода адресуется побайтово, буферный регистр данных клавиатуры может располагаться по адресу DEVADDR, а ее 8-разрядный регистр состояния — по адресу DEVADDR + 1.

Все 16-разрядное адресное пространство вмещает 64 Кбайт адресуемых единиц хранения. На него можно ссылаться через регистр DX посредством команды ввода

```
IN  REG,DX
```

где регистром REG может быть регистр AL или EAX. Здесь 16-разрядный адрес устройства содержится в регистре DX, представляющем 16 старших разрядов регистра EDX, а размер пересылаемого значения определяется размером операнда REG. Соответствующая команда вывода такова:

```
OUT  DX,REG
```

3.21.3. Блочная пересылка

В дополнение к командам IN и OUT, пересылающим между процессором и устройством ввода-вывода один информационный элемент, в архитектуре системы команд IA-32 определены две команды блочной пересылки: REPINS и REPOUTS. Они предназначены для последовательной передачи блоков данных между памятью и устройством ввода-вывода. Суффикс S в кодах этих операций означает string (строка), а префикс REP — repeat (повторять [пересылку элементов, пока не будет переслан весь блок данных]). Параметры, определяющие операцию пересылки, задаются не в командах REPINS и REPOUTS, а в регистрах DX (16-разрядный адрес устройства ввода-вывода), EDI (32-разрядный адрес начала блока в памяти) и ECX (количество пересылаемых элементов данных).

Суффикс B или D в мнемоническом обозначении кода операции определяет размер элемента — байт или двойное слово. Команда REPINSB пересылает блок байтов, а команда REPINSD — блок двойных слов. Команды блочной пересылки работают так: после пересылки каждого элемента данных значение индексного регистра EDI увеличивается на 1 или 4, в зависимости от размера элемента данных, а значение регистра ECX уменьшается на 1. Операции пересылки повторяются до тех пор, пока содержимое регистра-счетчика ECX не станет равным 0. Таким образом, каждая команда блочной пересылки эквивалентна целому программному циклу с регистром ECX в качестве счетчика цикла.

Для примера рассмотрим ситуацию, когда блок из 128 двойных слов пересылается с диска в основную память. Двойное слово данных содержится в адресуемом буферном регистре данных дискового устройства по адресу DISKDATA. Блок данных должен размещаться в памяти начиная с адреса MEMBLOCK. В регистр счетчика ECX необходимо записать начальное значение 128. Для выполнения пересылки можно применить последовательность команд

```
LEA     EDI,MEMBLOCK
MOV     DX,DISKDATA
MOV     ECX,128
REPINSD
```

В ней предполагается, что метка MEMBLOCK объявлена как адресная метка, а DISKDATA — как метка данных, определенная директивой EQU и представляющая 16-разрядный адрес буферного регистра данных устройства.

Теперь вы знаете, как выполняется изолированный ввод-вывод и как при помощи одной команды можно произвести пересылку блока, состоящего из большого количества элементов данных, с использованием регистров-счетчиков адреса (EDI) и элемента данных (ECX).

3.22. Подпрограммы

Как было отмечено в разделе 2.9, стек процессора удобно применять для выполнения операций, связанных со входом в подпрограммы и возвратом из таковых. В архитектуре IA-32 в качестве указателя стека используется регистр ESP, указывающий на текущую вершину стека процессора (то есть на его верхний элемент). Стек растет в направлении уменьшения адресов. Принцип его организации описан в разделе 2.8. Ширина стека составляет 32 разряда, а это означает, что его элементы являются двойными словами.

Существует четыре команды для проталкивания элементов в стек и выталкивания их из стека. Команда

```
PUSH  src
```

уменьшает значение ESP на 4, а затем сохраняет двойное слово по адресу src в памяти, который указан в ESP. Команда POP

```
POP  dst
```

выполняет обратную операцию: считывает из памяти двойное слово, на которое указывает ESP, то есть считывает из стека верхний элемент, а затем сохраняет его по адресу dst и увеличивает значение ESP на 4, удаляя тем самым верхний элемент из стека. Регистр ESP используется в этой команде неявно. Исходный и результирующий операнды задаются в одном из режимов адресации IA-32. Еще две команды предназначены для выталкивания из стека и проталкивания в него сразу нескольких элементов. Команда

```
PUSHAD
```

проталкивает в стек содержимое восьми регистров общего назначения, от EAX до EDI, а команда

```
POPAD
```

выталкивает их из стека в обратном порядке. При извлечении сохраненного значения ESP команда POPAD удаляет его из стека, не загружая в регистр ESP, и продолжает выталкивать последующие элементы, записывая их в соответствующие регистры. Применение этих двух команд при реализации подпрограмм позволяет более эффективно сохранять и восстанавливать содержимое всех регистров.

Приведенную на рис. 3.40, *а* программу можно переписать в виде подпрограммы, как показано на рис. 3.45. Параметры будут передаваться ей через регистры. Адрес первого из складываемых чисел, NUM1, загружается вызывающей программой в регистр EBX, а количество складываемых чисел, N, — в регистр ECX. В вызывающей программе предполагается, что сумма будет записана подпрограммой в регистр EAX. Таким образом, регистры EBX, ECX и EAX используются для

передачи параметров. Регистр EDI подпрограмма при выполнении сложения задействует в качестве индексного регистра, поэтому его содержимое должно сохраняться и восстанавливаться при помощи команд PUSH и POP.

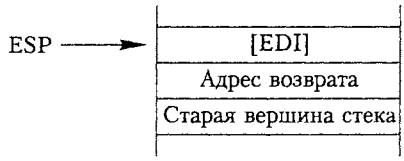
Подпрограмма вызывается командой

```
CALL LISTADD
```

Первым делом эта команда проталкивает в стек адрес возврата, а затем выполняет переход по адресу LISTADD. Содержимое стека после сохранения в нем содержимого регистра EDI показано на рис. 3.45, *б*. Адрес возврата в нашем примере — это адрес команды MOV, непосредственно следующей в вызывающей программе за командой CALL. Команда RET возвращает управление вызывающей программе, выталкивая из стека содержимое указателя команды EIP.

```
Вызывающая программа
                ⋮
                LEA     EBX,NUM1        Загрузка параметров
                MOV     ECX,N           в регистры EBX и ECX
                CALL    LISRADD         Переход к подпрограмме
                MOV     SUM,EAX         Сохранение суммы в памяти
                ⋮
Подпрограмма
LISTADD:        PUSH    EDI             Сохранение содержимого EDI
                MOV     EDI,0           EDI будет использоваться в качестве
                                        индексного регистра
                MOV     EAX,0           EAX будет использоваться для
                                        накопления суммы
STARTADD:       ADD     EAX, [EBX + EDI*4]  Прибавление следующего числа
                INC     EDI             Увеличение значения индекса
                DEC     ECX             Уменьшение значения счетчика
                JG      STARTADD        Если [ECX] > 0, выполняется переход
                                        к началу цикла
                POP     EDI             Восстановление значения EDI
                RET                     Возврат в вызывающую программу
```

а



б

Рис. 3.45. Программа с рис. 3.40, *а*, переписанная в виде подпрограммы для процессоров IA-32; параметры передаются через регистры: вызывающая программа и подпрограмма (*а*); содержимое стека после сохранения значения EDI в подпрограмме (*б*)

На рис. 3.46 показан еще один вариант этой же программы, в котором параметры передаются подпрограмме через стек.

(Вершина стека располагается на уровне 2, показанном на приведенном ниже рисунке.)

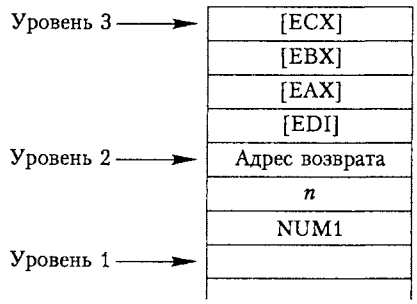
Вызывающая программа

```
        PUSH    OFFSET NUM1     Проталкивание параметров в стек
        PUSH    N
        CALL    LISTADD         Переход к подпрограмме
        ADD     ESP,4           Удаление счетчика N из стека
        POP     SUM             Выталкивание суммы в SUM
        :
```

Подпрограмма

```
LISTADD:PUSH    EDI             Сохранение EDI
        MOV     EDI,0           Этот регистр будет использоваться
                                в качестве индексного
        PUSH    EAX             Сохранение EAX
        MOV     EAX,0           Этот регистр будет использоваться
                                для накопления суммы
        PUSH    EBX             Сохранение EBX и загрузка
        MOV     EBX,[ESP+20]    адреса NUM1
        PUSH    ECX             Сохранение ECX и загрузка
        MOV     ECX,[ESP+20]    счетчика N
STARTADD:ADD    EAX,[EBX+EDI*4] Прибавление следующего числа
        INC     EDI             Увеличение значения индекса
        DEC     ECX             Уменьшение значения счетчика
        JG      STARTADD        Если сложение не завершено,
                                выполняется переход к началу цикла
        MOV     [ESP+24],EAX    На место NUM1 в стеке записывается
                                сумма
        POP     ECX             Восстановление значений регистров
        POP     EBX
        POP     EAX
        POP     EDI
        RET                     Возврат в вызывающую программу
```

а



б

Рис. 3.46. Программа с рис. 3.40, *а*, переписанная в виде подпрограммы для процессоров IA-32 (параметры передаются через стек): вызывающая программа и подпрограмма (*а*); содержимое стека после сохранения значения EDI в подпрограмме (*б*)

Параметры NUM1 и N проталкиваются в стек двумя командами PUSH в вызывающей программе. После выполнения команды CALL вершина стека распо-

лагается на уровне 2. Регистры EDI, EAX, EBX и ECX используются так же, как в подпрограмме на рис. 3.45. Их значения сохраняются в стеке, затем в них загружаются начальные значения и параметры. Эту работу выполняют первые 8 команд подпрограммы. В результате вершина стека оказывается на уровне 3. После сложения чисел при помощи цикла из четырех команд сумма помещается в стек на место параметра NUM1. Выполнив команду RET, команды вызывающей программы ADD и POP удаляют из стека параметр N и помещают результирующую сумму в память по адресу SUM, возвращая вершину стека на уровень 1.

Адрес	Команды		Комментарии
Вызывающая программа			
	⋮		
2000	PUSH	PARAM2	Помещение параметров в стек
2006	PUSH	PARAM1	
2012	CALL	SUB1	
2017	POP	RESULT	Сохранение результата
	ADD	ESP,4	Восстановление уровня стека
	⋮		
Первая подпрограмма			
2100 SUB1	PUSH	EBR	Сохранение регистра указателя на фрейм
	MOV	EBR,ESP	Загрузка указателя на фрейм
	PUSH	EAX	Сохранение регистров
	PUSH	EBX	
	PUSH	ECX	
	PUSH	EDX	
	MOV	EAX,[EBR+8]	Загрузка первого параметра
	MOV	EBX,[EBR+12]	Загрузка второго параметра
	⋮		
	PUSH	PARAM3	Помещение параметра в стек
2160	CALL	SUB2	
2165	POP	ECX	Выталкивание результата работы SUB2 в ECX
	⋮		
	MOV	[EBR+8],EDX	Помещение результата в стек
	POP	EDX	Восстановление регистров
	POP	ECX	
	POP	EBX	
	POP	EAX	
	POP	EBP	Восстановление регистра указателя на фрейм
	RET		Возврат в главную программу
Вторая подпрограмма			
3000 SUB2:	PUSH	EBR	Сохранение регистра указателя на фрейм
	MOV	EBP,ESP	
	PUSH	EAX	Загрузка указателя на фрейм
	PUSH	EBX	Сохранение регистров
	MOV	EAX,[EBP+8]	Загрузка параметров
	⋮		
	MOV	[EBP+8],EBX	Помещение результата SUB2 в стек
	POP	EBX	Восстановление регистров
	POP	EAX	
	POP	EBP	Восстановление регистра указателя на фрейм
	RET		Возврат в первую подпрограмму

Рис. 3.47. Вложенные подпрограммы на языке ассемблера IA-32

В завершение темы вызова подпрограмм мы рассмотрим пример обработки вложенных вызовов. На рис. 3.47 приведен код программы на языке ассемблера IA-32, эквивалентной программе, представленной на рис. 2.28. Стековые фреймы обоих подпрограммы вы видите на рис. 3.48. Указателем на фрейм служит регистр EBP. Структура вызывающей программы и подпрограмм очень близка к структуре кода на рис. 2.28. Однако существенным отличием кода для IA-32 является то, что для сохранения и восстановления регистров в нем применяются последовательности команд PUSH и POP, а не команды MoveMultiple. В наборе команд IA-32 имеются команды PUSHAD и POPAD, с помощью которых можно сохранить в стеке и восстановить из него все восемь регистров общего назначения, но в программе на рис. 3.47 мы предпочли воспользоваться отдельными командами PUSH и POP, поскольку в подпрограммах задействована только половина всех регистров.

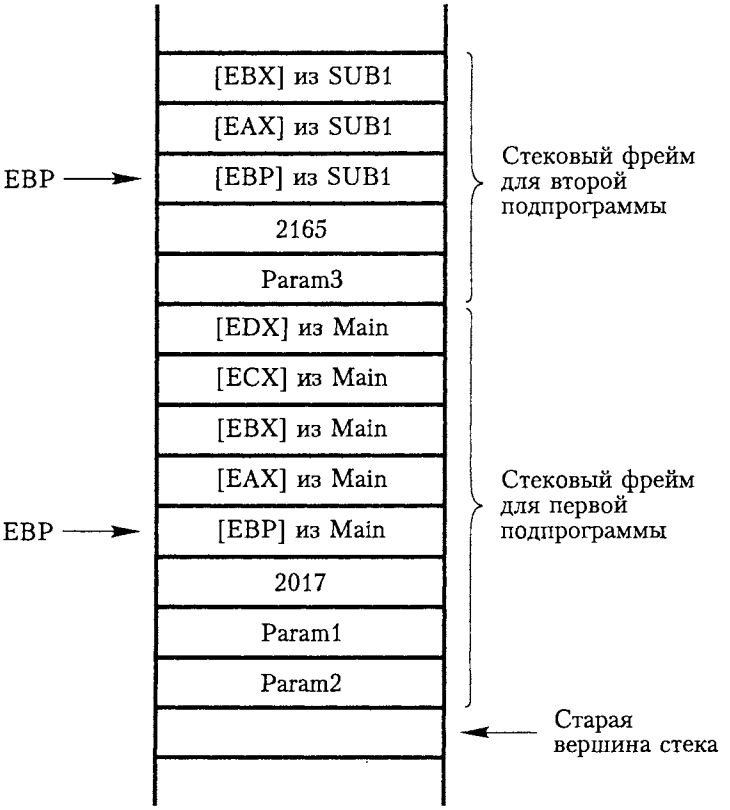


Рис. 3.48. Стековые фреймы для программы, представленной на рис 3.47

3.23. Другие команды

Мы рассмотрели лишь малую часть набора команд архитектуры IA-32. Еще несколько важных команд будут представлены ниже.

3.23.1. Команды умножения и деления

Кроме команд для сложения и вычитания целых чисел со знаком, описанных в разделе 3.17, в наборе команд IA-32 имеются команды для целочисленного умножения и деления, а также для выполнения арифметических операций над числами с плавающей запятой.

В общем случае в результате умножения двух 32-разрядных чисел получается произведение двойной длины, то есть 64-разрядное значение. Однако для многих приложений достаточно иметь результат одинарной длины, то есть 32-разрядное значение. В подобных ситуациях используются разные команды. Результат одинарной длины генерирует команда

```
IMUL  REG,src
```

помещающая 32-разрядное значение в регистр общего назначения REG. Исходный операнд может находиться либо в регистре, либо в памяти. В случае 64-разрядного результата команда

```
IMUL  src
```

использует в качестве второго операнда регистр EAX. Исходный операнд может располагаться либо в регистре, либо в памяти. Произведение двойной длины помещается в два регистра: старшая половина разрядов результата сохраняется в регистре EDX, а младшая — в регистре EAX.

Команда для выполнения целочисленного деления имеет следующий формат:

```
IDIV  src
```

Исходный операнд src является делителем. Делимое всегда находится в регистре EAX. Перед выполнением команды знак делимого должен быть расширен на регистр EDX. Для этого используется команда CDQ (конвертирование двойного слова в четверное), которая не имеет операндов, поскольку всегда выполняется над регистрами EAX и EDX.

Числа с плавающей запятой, о которых достаточно подробно рассказывается в главе 6, имеют гораздо больший диапазон значений, чем целые числа, и используются в первую очередь для научных вычислений. В архитектуре IA-32 применяется полный набор арифметических операций с такими числами. Их операнды и результаты располагаются в регистрах с плавающей запятой, показанных на рис. 3.37. Поддерживаются два формата чисел: с одинарной (32 разряда) и двойной (64 разряда) точностью.

3.23.2. Команды мультимедийного расширения

Двухмерные графические и видеоизображения можно представить массивом, состоящим из большого количества точек. Цвет и яркость каждой точки, называемой

пикселом, могут быть закодированы 8-битовым элементом данных. Обработка таких данных имеет две особенности. Во-первых, при обработке отдельных пикселов часто приходится выполнять очень простые арифметические и логические операции. Во-вторых, для некоторых приложений реального времени требуется исключительно высокая скорость вычислений. Сказанное касается и обработки аудиосигналов и речи, представляемых последовательностью отсчетов непрерывного аналогового сигнала, генерируемых через фиксированные промежутки времени.

Ускорить процесс обработки данных в приложениях обоих типов можно за счет разделения последовательности элементов данных (как правило, байтов или 16-разрядных слов) на маленькие группы, которые можно обрабатывать параллельно. В наборе команд IA-32 имеется множество таких, которые параллельно обрабатывают данные группами по 64 бита, называемыми четверными словами. (Четверное слово содержит восемь байт или четыре 16-разрядных слова.) Эти команды называются *MMX-командами* или *командами мультимедийного расширения* (multimedia extension). Их операнды могут располагаться в памяти или в регистрах с плавающей запятой. Таким образом, эти регистры служат двойной цели: в них могут храниться числа с плавающей запятой или операнды MMX-команд. При использовании в командах MMX регистры обозначаются как MM0–MM7.

Для пересылки 64-разрядных MMX-операндов между памятью и регистрами MMX используются специальные команды Move.

Команда

PADDB MMi,src

складывает соответствующие байты двух 8-байтовых операндов и помещает в регистр назначения восемь сумм. Исходный операнд может располагаться в памяти или в регистре MMX, но операнд назначения обязательно должен находиться в регистре MMX. Подобные команды имеются как для операций по вычитанию, так и для логических операций.

Типичной операцией, выполняемой в приложениях обработки сигналов, является свертка — умножение короткой последовательности отсчетов входных сигналов на константы, называемые *весовыми коэффициентами*, и сложение произведений для получения значения выходного сигнала. Подобные операции осуществляются с помощью специальной команды MMX, объединяющей умножение и сложение. В ней используются 64-разрядные операнды MMX, содержащие по четыре 16-разрядных элемента данных, представляющих отсчеты сигнала.

3.23.3. Векторные команды

В архитектуре IA-32 определен набор команд, предназначенных для выполнения арифметических операций над маленькими группами чисел с плавающей запятой. Команды SIMD (Single Instruction, Multiple Data — одиночный поток команд и множественный поток данных) полезны для векторных и матричных вычислений в научных приложениях. В терминологии Intel они называются командами *потокового расширения SIMD* (Streaming SIMD Extension, *SSE*). Эти команды обрабатывают составные операнды длиной 128 бит, состоящие из четырех 32-разрядных чисел с плавающей запятой. Для хранения этих операндов имеются

128-разрядные регистры (на рис. 3.37 не показаны). Двумя базовыми командами этой группы являются команды сложения и умножения. Они воздействуют на четыре соответствующие пары исходных 32-разрядных значений, которые находятся в составных 128-разрядных операндах, и помещают четыре отдельных результата в 128-разрядный операнд назначения.

3.24. Примеры программ

В этом разделе приводятся примеры программ для архитектуры IA-32, рассмотренных в разделе 2.11.

3.24.1. Программа для вычисления скалярного произведения двух векторов

На рис. 3.49 приведена программа вычисления скалярного произведения двух векторов для процессоров архитектуры IA-32. Начальные адреса этих векторов равны AVEC и BVEC. Программа написана по образцу программы, показанной на рис. 2.33. В программе на рис. 3.49 для доступа к последовательным элементам векторов используется базовая индексная адресация. В качестве индексного регистра применяется регистр EDI. Коэффициент масштабирования равен 4, поскольку элементы векторов являются двойными словами (4 байта). В качестве счетчика цикла используется регистр ECX, инициализированный значением *n*. Это позволяет задействовать команду LOOP (см. раздел 3.17 и рис. 3.40, *б*), которая сначала уменьшает значение регистра ECX, а затем выполняет условный переход по адресу LOOPSTART, если содержимое регистра ECX не равно нулю. Предполагается, что произведение двух элементов векторов поместится в двойное слово, поэтому в команде умножения IMUL явно задан регистр назначения EDX (см. раздел 3.23).

```
            LEA     EBP,AVEC        EBP указывает на вектор A
            LEA     EBX,BVEC        EBX указывает на вектор B
            MOV     ECX,N           ECX используется в качестве счетчика
                                    цикла
            MOV     EAX,0           В EAX накапливается сумма
                                    произведений
            MOV     EDI,0           EDI выполняет функции индексного
                                    регистра
LOOPSTART:  MOV     EDX,[EBP+EDI*4] Вычисление произведения
            IMUL    EDX,[EBX+EDI*4] следующих компонентов
            INC     EDI             Приращение индекса
            ADD     EAX,EDX         Прибавление произведения
                                    к предыдущей сумме
            LOOP    LOOPSTART       Переход к началу цикла, если
                                    перемножены еще не все компоненты
            MOV     DOTPROD,EAX     Сохранение скалярного произведения
                                    в памяти
```

Рис. 3.49. Программа для процессоров IA-32, вычисляющая скалярное произведение двух векторов

3.24.2. Программа сортировки байтов

На рис. 3.50, *б* вы видите программу для процессора IA-32, выполняющую сортировку байтов. Она написана по образцу программы, приведенной на рис. 2.34, *б*. Регистр EAX инициализируется значением LIST и используется в качестве базового регистра для доступа к элементам списка при базовой индексной адресации. Регистр EDI служит индексным регистром для внешнего цикла (индекс *j*), а регистр ECX — индексным регистром для внутреннего цикла (индекс *k*). В регистре DL содержится текущий наибольший байт сортируемого подсписка. Программа на рис. 3.50, *б* очень похожа на представленную на рис. 2.34, *б*, с той лишь разницей, что в ней элементы LIST(*k*) и LIST(*j*) меняются местами при помощи одной команды — XCHG, тогда как второй программе для этой цели требуются целые три команды, а также регистр для временного хранения данных.

```
int main(int argc, char* argv[])
{
   for (j = n-1; j>0; j = j-1)
   {for k = j-1; k>=0; k = k-1)
      {if (LIST[k] > LIST[j])
         {TEMP = LIST[k];
          LIST[k] = LIST[j];
          LIST[j] = TEMP;
         }
      }
   }
```

а

```
        LEA     EAX,LIST    Загрузка указателя на список в базовый
        MOV     EDI,N       регистр (EAX) и инициализация индексного
        DEC     EDI         регистра внешнего цикла (EDI) значением
                            j = n - 1
OUTER:  MOV     ECX,EDI     Загрузка в индексный регистр внутреннего
        DEC     ECX         цикла (ECX) значения k = j - 1
        MOV     DL,[EAX+EDI] Загрузка значения LIST(j) в регистр DL
INNER:  CMP     [EAX+ECX],DL Сравнение LIST(k) и LIST(j)
        JLE     NEXT        Если LIST(k) ≤ LIST(j), увеличение индекса k
                            на 1 и переход к новому элементу LIST(k)
        XCHG    [EAX+ECX],DL В противном случае LIST(k) и LIST(j)
                            меняются местами, а в регистр DL помещается
        MOV     [EAX+EDI],DL новое значение LIST(j)
NEXT:   DEC     ECX         Уменьшение значения индекса внутреннего
                            цикла k
        JGE     INNER       Повторное выполнение внутреннего цикла или
                            выход из него
        DEC     EDI         Уменьшение значения индекса внешнего
                            цикла j
        JG      OUTER       Повторное выполнение внешнего цикла или
                            выход из него
```

б

Рис. 3.50. Программа сортировки байтов для процессоров IA-32: на языке C (*а*); на языке ассемблера (*б*)

3.24.3. Подпрограммы для вставки и удаления элементов связного списка

Программы на рис. 3.51 и 3.52, выполняющие вставку и удаление элементов связного списка, очень близки к программам, представленным на рис. 2.37 и 2.38. Параметры им передаются через регистры. Причем регистры с именами RHEAD, RNEWREC, RIDNUM, RCURRENT и RNEXT используются так же, как в универсальных подпрограммах. Указанные имена применяются вместо имен регистров IA-32 EAX, EBX и т. д. Для хранения кода новой вставляемой записи задействован шестой регистр, RNEWID, в который первая команда подпрограммы с рис. 3.51 загружает код новой записи. Остальная часть подпрограммы один к одному повторяет команды подпрограммы, приведенной на рис. 2.37. Подпрограмма для удаления записей, которую вы видите на рис. 3.52, также полностью повторяет команды подпрограммы с рис. 2.38.

Подпрограмма

```
            MOV   RNEWID,[RNEWREC]
            CMP   RHEAD,0          Проверка того, пуст ли список
            JG    HEAD
            MOV   RHEAD,RNEWREC    Если список пуст, новая
                                   запись становится его
            RET                    единственным элементом
HEAD:       CMP   RNEWID,[RHEAD]   Проверка того, должна ли
                                   новая запись стать первой
            JG    SEARCH
            MOV   [RNEWREC+4],RHEAD Если да, делаем ее
            MOV   RHEAD,RNEWREC    первой записью списка
            RET
SEARCH:     MOV   RCURRENT,RHEAD   В противном случае
                                   с использованием
LOOPSTART:  MOV   RNEXT,[RCURRENT+4] регистров RCURRENT
            CMP   RNEXT,0          и RNEXT производится
            JE    TAIL             перемещение по
            CMP   RNEWID,[RNEXT]   списку в поисках
            JL    INSERT           места вставки новой записи
            MOV   RCURRENT,RNEXT
            JMP   LOOPSTART
INSERT:     MOV   [RNEWREC+4],RNEX
TAIL:       MOV   [RCURRENT+4],RNEWREC
            RET
```

Рис. 3.51. Подпрограмма для процессоров IA-32, вставляющая в связный список новый элемент

```
Подпрограмма
DELETION:   CMP   RIDNUM,[RHEAD]      Проверка того, является ли
                                      удаляемая запись первой
            JG    SEARCH
            MOV   RHEAD,[RHEAD+4]     Если да, удаляем ее
            RET
SEARCH:     MOV   RCURRENT,RHEAD      В противном случае
                                      с использованием
LOOPSTART:  MOV   RNEXT,[RCURRENT+4]  регистров RCURRENT
            CMP   RIDNUM,[RNEXT]      и RNEXT перемещаемся
            JE    DELETE              по списку в поисках
            MOV   RCURRENT,RNEXT      удаляемой записи
            JMP   LOOPSTART
DELETE:     MOV   RTEMP,[RNEXT+4]
            MOV   [RCURRENT+4],RTEMP
            RET
```

Рис. 3.52. Подпрограмма для процессоров IA-32, удаляющая элемент из связного списка

Как и в программах на рис. 2.37 и 2.38, в программе вставки в связный список нового элемента для процессоров IA-32 предполагается, что код новой записи не совпадает ни с одним из кодов, имеющихся с списке записей, а в программе удаления предполагается, что в списке имеется запись с кодом, заданным в регистре RIDNUM. В упражнениях 3.72 и 3.73 читателю дается задание подумать, как изменить эти подпрограммы таким образом, чтобы в случае ошибочности указанных предположений выдавалось сообщение об ошибке.

3.25. Резюме

В настоящей главе были рассмотрены системы команд трех типов процессоров – ARM, Motorola 68000 и Intel IA-32. Система команд процессора ARM, как вы помните, основана на архитектуре RISC, а системы команд 68000 и IA-32 — на архитектуре CISC, причем IA-32 считается одним из ее наиболее ярко выраженных «представителей».

Все команды процессоров ARM кодируются одним 32-разрядным словом. Операции могут выполняться только над данными, содержащимися в регистрах процессора. Для пересылки операндов между регистрами процессора и памятью применяются команды Load и Store. Мы показали, как благодаря возможности условного выполнения всех команд и разнообразию способов сдвига операндов можно наиболее эффективно реализовать типичные программные задачи.

Машинные команды процессора архитектуры 68000 имеют переменную длину, которая зависит от сложности выполняемой операции и количества информации, необходимой для формирования исполнительного адреса операнда в памяти. В отдельных командах могут использоваться и регистры, и данные, хранящиеся в памяти.

В архитектуре IA-32 реализован широчайший набор команд для выполнения самых разных операций над различными типами данных.

Упражнения

Процессоры ARM

3.1. В регистрах и памяти компьютера ARM хранится следующая информация.

Регистр R0 содержит значение 1000.
Регистр R1 содержит значение 2000.
Регистр R2 содержит значение 1016.
Регистр R6 содержит значение 20.
Регистр R7 содержит значение 30.

Числа 1, 2, 3, 4, 5 и 6 содержатся в последовательных словах памяти начиная с адреса 1000. Каким будет результат выполнения каждого из трех следующих блоков команд при указанных начальных значениях?

```
а) LDR     R8,[R0]
   LDR     R9,[R0,#4]
   LDR     R10,R8,R9
б) LDR     R8,[R0]
   STR     R6,[R1,#-4]!
   STR     R7,[R1,#-4]!
   LDR     R8,[R1],#
   LDR     R9,[R1],#
   SUB     R10,R8,R9
в) LDMIA   R2!,{R4,R5}
   ADD     R4,R4,R5
```

3.2. Для каких из перечисленных ниже команд ARM ассемблер выдаст сообщение о синтаксической ошибке: Почему?

```
а) ADD     R2,R2,R2
б) SUB     R0,R1,[R2,#4]
в) MOV     R0,#2,1010101
г) MOV     R0,#257
д) ADD     R0,R1,R11,LSL #8
е) ADD     R4,R4,R5
```

3.3. В случае, когда в регистр процессора ARM посредством команды Load из памяти загружается 1 байт, старшие 24 разряда регистра заполняются нулями (см. раздел 3.1.12). Если загруженный байт представляет 8-разрядное целое число со знаком в форме дополнения до двух, перед использованием

в арифметических операциях его знак должен быть расширен до 32 разрядов. Предположим, что такой байт загружен в регистр R0. Напишите короткую программу для расширения знака числа до 32 разрядов, то есть на всю длину регистра. (Подсказка: для копирования содержимого регистра R0 после выполнения одной из операций сдвига, LSL, LSR или ASR (см. раздел 2.10.2), обратно в R0 используйте команду пересылки.)

3.4. Напишите программу для процессора ARM, которая изменяла бы порядок битов в регистре R2 на обратный. Например, если первоначально в регистре R2 хранится значение 1110 ... 0100, то после выполнения вашей программы в нем должно содержаться значение 0010...0111. (Подсказка: используйте операции сдвига и циклического сдвига.)

3.5. В результате *трассировки* программы может быть получен листинг содержимого некоторых регистров и адресов памяти в указанные моменты выполнения программы. Как будет каждый раз изменяться содержимое регистров R0, R1 и R2 после трехкратного выполнения команды BGT в программе, представленной на рис. 3.7? Приведите результаты в виде таблицы с именами трех регистров в заголовках столбцов. Содержимое регистров после каждого выполнения команды BGT укажите в трех строках таблицы. Данные для программы приведены на рис. 3.8.

3.6. Напишите программу для процессора ARM, сравнивающую значения соответствующих байтов из двух списков байтов и помещающую большее из значений в третий список. Два исходных списка начинаются по адресам X и Y, а результирующий список — по адресу LARGER. Длина списка хранится в памяти по адресу N.

3.7. Напишите программу для процессора ARM, которая бы выполняла задачу, связанную с обработкой символов. Известно, что строка из n символов хранится в памяти в виде последовательности байтов начиная с адреса STRING. Еще одна, более короткая строка из m символов хранится в памяти по адресу SUBSTRING. Программа должна просмотреть строку, начинающуюся по адресу STRING, и определить, содержится ли в ней непрерывная подстрока, идентичная строке по адресу SUBSTRING. Параметры n и m ($n > m$) хранятся в памяти по адресам N и M соответственно. Если подстрока найдена, в регистр R0 должен быть записан ее адрес. В противном случае регистр R0 должен быть очищен (установлен в 0). Многократные вхождения строки программа выявлять не должна — нужен лишь адрес первого вхождения подстроки в строку.

3.8. Напишите программу для процессора ARM, генерирующую первые n чисел последовательности Фибоначчи. В этой последовательности первыми двумя числами являются 0 и 1, а каждое следующее число генерируется путем сложения двух предшествующих. Например, для $n = 8$ последовательность Фибоначчи такова:

0, 1, 1, 3, 3, 5, 8, 13

Ваша программа должна записать эти числа в последовательные слова памяти начиная с адреса MEMLOC. Значение n хранится по адресу N.

3.9. Напишите программу для процессора ARM, выполняющую преобразование слова текста, набранного в нижнем регистре, в слово в верхнем регистре. Слово состоит из символов ASCII, хранящихся в памяти в последовательных байтах начиная с адреса WORD, и заканчивается символом пробела. (Сведения об ASCII-кодах представлены в приложении Д.)

3.10. Список полученных студентами оценок, приведенный на рис. 2.14, изменился, и теперь для каждого студента в нем содержится j оценок. Количество студентов равно n. Напишите программу для процессора ARM, вычисляющую сумму оценок по каждому из тестов и записывающую эти суммы в слова памяти по адресам SUM, SUM + 4, SUM + 8,.... Количество тестов j больше количества регистров процессора, так что для суммирования оценок не может использоваться программа, аналогичная приведенной на рис. 2.15. Воспользуйтесь вложенным циклом, как предлагалось в разделе 2.5.3. Во внутреннем цикле должна накапливаться сумма по одному тесту, а во внешнем — выполняться проход по всем тестам. Количество тестов j хранится в памяти по адресу J, который меньше адреса N.

3.11. Рассмотрим массив чисел A(i,j), где индекс строки i принимает значения от 0 до $n - 1$, а индекс столбца j — значения от 0 до $m - 1$. Данный массив хранится в памяти компьютера ARM построчно, а элементы каждой строки занимают m последовательных слов. Напишите подпрограмму для процессора ARM, предназначенную для поэлементного сложения столбцов x и y с записью сумм в столбец y. Индексы x и y передаются подпрограмме в регистрах R1 и R2, параметры n и m — в регистрах R3 и R4, а адрес элемента A(0,0) — в регистре R0. Допускается применение любых режимов адресации из табл. 3.1.

3.12. Напишите подпрограмму для процессора ARM, считывающую с клавиатуры n символов, помещающую их по мере ввода в пользовательский стек и отображающую их на дисплее. В качестве указателя стека используйте регистр R6. Информация о количестве символов n содержится в памяти по адресу N.

3.13. Предположим, что среднее время выборки и выполнения команды программы, приведенной на рис. 3.9, составляет 20 нс. Сколько раз для каждого из введенных символов будет выполнена команда BEQ READ, если символы вводятся в клавиатуры со скоростью 10 символов в секунду? Время, уходящее на вывод символа на экран, значительно меньше времени, проходящего между вводом последовательных символов с клавиатуры.

3.14. В программе для процессора ARM, приведенной на рис. 3.9, чтение и вывод символов выполняется без вызова соответствующих подпрограмм. Перепишите эту программу таким образом, чтобы она состояла из главной программы, вызывающей подпрограмму GETCHAR для ввода одного символа и подпрограмму PUTCHAR для вывода одного символа. Адрес регистра INSTATUS передается подпрограмме GETCHAR в регистре R1, а считанный с клавиатуры символ главная подпрограмма получает через регистр R3. Адрес регистра OUTSTATUS и подлежащий выводу символ передаются подпрограмме PUTCHAR в регистрах R2 и R3 соответственно. Любые

другие используемые в подпрограммах регистры должны сохраняться и восстанавливаться с использованием стека, указатель на который находится в регистре R13. Запись символов в память и проверка на наличие символа конца строки CR должна выполняться главной программой.

3.15. Повторите упражнение 3.14 с передачей параметров через стек.

3.16. Напишите программу для процессора ARM, выполняющую ввод с клавиатуры трех цифр. Каждая цифра представлена ASCII-кодом (см. приложение Д). Вместе эти три цифры составляют десятичное целое число из диапазона от 0 до 999. Сначала вводится старшая цифра. Преобразуйте это число в двоичное представление. Для упрощения преобразования в памяти хранятся две таблицы слов, содержащие по 10 элементов. Первая из них начинается по адресу TENS. В ней содержатся двоичные представления десятичных значений 0, 10, 20, ..., 90. Вторая таблица начинается по адресу HUNDREDS и содержит двоичные представления десятичных значений 0, 100, 200, ..., 900.

3.17. Программа для преобразования из десятичного формата в двоичный, которую вы написали в упражнении 3.16, должна содержать две вложенные подпрограммы. Главная подпрограмма вызывает первую подпрограмму и передает ей два параметра через стек, указатель на который находится в регистре R13. Первым параметром является адрес 3-байтового буфера памяти для хранения введенных десятичных цифр. Второй параметр — это адрес, по которому должно быть записано результирующее двоичное значение. Первая подпрограмма считывает с клавиатуры три символа и вызывает вторую подпрограмму для выполнения преобразования. На этот раз параметры передаются через регистры процессора. Обе подпрограммы должны сохранять содержимое используемых ими регистров в стеке.

а) Напишите две указанные подпрограммы на языке ассемблера процессора ARM.

б) Приведите содержимое стека сразу после выполнения команды вызова второй подпрограммы.

3.18. Используя очередь, о которой речь шла в упражнении 2.18, напишите подпрограммы APPEND и REMOVE для процессора APM, выполняющие пересылку данных между регистрами процессора и очередью. Будьте внимательны при анализе и обновлении состояния очереди и указателей.

3.19. Приведите результаты трассировки программы сортировки байтов, показанной на рис. 3.15, *б* в формате, описанном в упражнении 3.5. Покажите содержимое регистров R0, R2 и R3 и последовательности из 5 байт, расположенных по адресам LIST, LIST + 1, ..., LIST + 4, после каждого выполнения последней команды программы. Предполагается, что LIST = 1000, начальными значениями байтов являются 120, 13, 106, 45 и 67, а [LIST] = 120.

3.20. Перепишите приведенную на рис. 3.15, *б* программу сортировки байтов в виде подпрограммы, сортирующей список последовательно расположенных в памяти 32-разрядных положительных целых чисел. Вызывающая програм-

ма должна передавать подпрограмме адрес этого списка. Первое 32-разрядное число по указанному адресу определяет количество элементов списка, за которым следуют сортируемые значения.

3.21. Рассмотрим программу сортировки байтов, приведенную на рис. 3.15, *б*. На каждом проходе по подсписку от LIST(j) до LIST(0) все элементы списка, для которых LIST(k) > LIST(j), меняются местами. В качестве альтернативы можно отслеживать адрес наибольшего значения в подсписке и в конце просмотра выполнять не более одного обмена. Перепишите программу с использованием второго подхода. Каковы его преимущества?

3.22. Предположим, что список полученных студентами оценок, представленный на рис. 2.14, хранится в памяти в виде связного списка (рис. 2.36). Напишите программу для процессора ARM, выполняющую ту же задачу, что и программа на рис. 2.15. Первая запись списка хранится в памяти по адресу 1000.

3.23. Приведенная на рис. 3.16 подпрограмма вставки записи в связный список не проверяет код этой записи на предмет совпадения с кодами других записей, уже имеющихся в списке. А что произойдет в случае совпадения? Модифицируйте подпрограмму таким образом, чтобы в указанном случае она возвращала в регистре R10 адрес записи, код которой совпадает с кодом вводимой записи, или 0, если вставка выполнена успешно.

3.24. Подпрограмма удаления записи из связного списка (рис. 3.17) не проверяет наличия в списке записи с кодом, указанным в регистре RIDNUM. Модифицируйте эту программу таким образом, чтобы в случае успешного удаления она возвращала в регистре RIDNUM значение 0 или оставляла содержимое этого регистра неизменным, если запись в списке не найдена.

Процессор Motorola 68000

3.25. Рассмотрим следующее состояние процессора 68000.

Регистр D0 содержит значение $1000.
Регистр A0 содержит значение $2000.
Регистр A1 содержит значение $1000.
В памяти по адресу $1000 содержится $2000.
В памяти по адресу $2000 содержится $3000.

Каким будет результат выполнения каждой из трех перечисленных ниже команд при указанных начальных значениях? Сколько байтов занимает каждая команда? Сколько обращений к памяти требуется на выборку и выполнение каждой из команд?

а) ADD.L D0,(A0)
б) ADD.L (A1,D0),D0
в) ADD.L #$2000,(A0)

3.26. Найдите синтаксические ошибки в следующих командах процессора 68000:

а) ADDX -(A2),D3

```
б) LSR.L    #9,D2
в) MOVE.B   520(A2,D2)
г) SUBA.L   12(A2,PC),A0
д) CMP.B    #254,$12(A2,D1.B)
```

3.27. В результате *трассировки* программы может быть получен листинг содержимого определенных регистров и адресов памяти в указанные моменты выполнения программы. Каким будет содержимое регистров D0, D1 и A2, а также адресов памяти N, NUM1 и SUM каждый раз после пятикратного выполнения команды ADD.W и после выполнения последней команды MOVE.L в программе на рис. 3.25. Приведите результаты в виде таблицы с именами трех регистров и адресами памяти в заголовках столбцов. Содержимое регистров и памяти после каждого выполнения указанных команд укажите в шести строках таблицы. Начальные значения, с которыми работает программа, таковы: [SUM] = 0, [N] = 5, NUM1 = 2400, а пять чисел — это 83, 45, 156, −250 и 100.

3.28. Рассмотрим следующую программу для процессора 68000:

```
        MOVEA.L   MEM1,A0
        MOVEA.L   MEM2,A2
        ADDA.L    A0,A1
        MOVEA.L   A0,A2
        MOVE.B    (A0)+,D0
LOOP    CMP.B     (A0)+,D0
        BLE       NXT
        LEA       −1(A0),A2
        MOVE.B    (A2),D0
NXT     CMPA.L    A0,A1
        BGT       LOOP
        MOVE.L    A2,DESIRED
```

а) Что делает эта программа?

б) Сколько 16-разрядных слов необходимо для хранения данной программы в памяти?

в) Приведите выражение для определения необходимого количества обращений к памяти. Данное выражение должно иметь вид $T = a + bn + cm$, где n — это количество итераций цикла, m — количество невыполненных переходов по адресу NXT, а a, b и c — константы.

3.29. Рассмотрим программы для процессора 68000, приведенные на рис. У3.1.

а) Оставляют ли эти программы исходное значение по адресу RSLT?

б) Какие задачи они выполняют?

в) Сколько байтов памяти необходимо для хранения каждой из программ?

г) Какая программа чаще обращается к памяти?

д) Каковы преимущества и недостатки этих программ?

Программа 1			Программа 2		
	CLR.L	D0		MOVE.W	#$FFFF,D0
	MOVEA.L	#LIST,A0		MOVEA.L	#LIST,A0
LOOP	MOVE.W	(A0)+,D1	LOOP	LSL.W	(A0)+
	BGE	LOOP		BCC	LOOP
	ADDQ.L	#1,D0		LSL.W	#1,D0
	CMPI	#17,D0		BCS	LOOP
	BLT	LOOP		MOVE.W	−2(A0),RSLT
	MOVE.W	−2(A0),RSLT			

Рис. УЗ.1. Две программы для процессора 68000

3.30. Напишите программу для процессора 68000, которая сравнивает значения соответствующих байтов из двух списков байтов и помещает большее из значений в третий список. Два исходных списка начинаются по адресам X и Y, а результирующий список — по адресу LARGER. Длина списка хранится в памяти по адресу N.

3.31. Напишите программу для процессора 68000, которая выполняла бы ряд задач, связанных с обработкой символов. Строка из n символов хранится в памяти в виде последовательности байтов начиная с адреса STRING. Еще одна, более короткая строка из m символов хранится в памяти по адресу SUBSTRING. Программа должна просмотреть строку, начинающуюся по адресу STRING, чтобы определить, содержится ли в ней непрерывная подстрока, идентичная строке по адресу SUBSTRING. Параметры n и m, где $n > m$, хранятся в памяти соответственно по адресам N и M. Если подстрока найдена, в регистр D0 должен быть записан ее адрес. В противном случае регистр D0 должен быть очищен (установлен в 0). Многократные вхождения строки программа выявлять не должна — нужен только адрес первого вхождения подстроки в строку.

3.32. Напишите программу для процессора 68000, генерирующую первые n чисел последовательности Фибоначчи. В этой последовательности первыми двумя числами являются 0 и 1, а каждое следующее число генерируется путем сложения двух предшествующих чисел. Например, для $n = 8$ последовательность Фибоначчи такова:

0, 1, 1, 3, 3, 5, 8, 13

Ваша программа должна записать эти числа в последовательные слова памяти начиная с адреса MEMLOC. Значение n хранится по адресу N. Каково максимальное значение n, допустимое для вашей программы?

3.33. Напишите программу для процессора 68000, выполняющую преобразование слова текста, набранного в нижнем регистре, в слово в верхнем регистре. Слово состоит из символов ASCII, хранящихся в памяти в последовательных байтах начиная с адреса WORD, и заканчивается символом пробела. (Сведения об ASCII-кодах представлены в приложении Д.)

3.34. Список полученных студентами оценок, приведенный на рис. 2.14, изменился, и теперь для каждого студента в нем содержится j оценок. Каждый элемент в списке является 16-разрядным словом, так что значение счетчика каждый раз должно увеличиваться на 2. Общее количество студентов составляет n. Напишите программу для процессора 68000, вычисляющую сумму оценок по каждому из тестов и записывающую эти суммы в слова памяти по адресам SUM, SUM + 4, SUM + 8, ... Количество тестов j больше количества регистров процессора, так что для суммирования оценок не может использоваться программа, аналогичная приведенной на рис. 2.15. Воспользуйтесь вложенным циклом, как предлагалось в разделе 2.5.3. Во внутреннем цикле должна накапливаться сумма по одному тесту, а во внешнем — выполняться проход по всем тестам. Количество тестов j хранится в памяти по адресу J, который меньше адреса N.

3.35. Напишите подпрограмму для процессора 68000, считывающую с клавиатуры n символов, помещающую их по мере ввода в пользовательский стек, а затем отображающую их на дисплее. В качестве указателя стека используйте регистр A0. Информация о количестве символов содержится в памяти по адресу N.

3.36. Предположим, что среднее время выборки и выполнения команды программы, приведенной на рис. 3.27, составляет 20 нс. Если символы вводятся с клавиатуры со скоростью 10 символов в секунду, сколько раз для каждого из них будет выполнена команда BEQ READ? Время, уходящее на вывод символа на экран, много меньше времени, проходящего между вводом последовательных символов с клавиатуры.

3.37. В программе для процессора 68000, приведенной на рис. 3.27, чтение и вывод символов выполняется без вызова подпрограмм. Перепишите эту программу таким образом, чтобы она состояла из главной программы, вызывающей для ввода одного символа подпрограмму GETCHAR, и подпрограммы для вывода одного символа PUTCHAR. Адреса регистров INSTATUS и DATAIN передаются подпрограмме GETCHAR в регистрах A0 и A1, а считанный с клавиатуры символ главная подпрограмма получает через регистр D0. Адреса регистров OUTSTATUS и DATAOUT и подлежащий выводу символ передаются подпрограмме PUTCHAR в регистрах A2, A3 и D0 соответственно. Любые другие используемые в подпрограммах регистры должны сохраняться и восстанавливаться с применением стека процессора, указатель на который находится в регистре A7. Запись символов в память и проверка на наличие символа конца строки CR должна выполняться главной программой.

3.38. Повторите упражнение 3.37, передавая параметры через стек.

3.39. Используя очередь, описанную в упражнении 2.18, напишите подпрограммы APPEND и REMOVE для процессора 68000, выполняющие пересылку данных между регистрами процессора и очередью. Будьте внимательны при анализе и обновлении состояния очереди и указателей.

3.40. Напишите программу для процессора 68000, выполняющую ввод с клавиатуры трех цифр. Каждая цифра представлена ASCII-кодом (см. приложение Д). Вместе эти три цифры составляют десятичное целое число из диапазона от 0 до 999. Первой вводится старшая цифра. Преобразуйте данное число в двоичное представление. Для того чтобы упростить процесс преобразования, воспользуйтесь хранящимися в памяти двумя таблицами слов, содержащими по десять элементов. Первая из них начинается по адресу TENS. В ней содержатся двоичные представления десятичных значений 0, 10, 20, ..., 90. Вторая таблица начинается по адресу HUNDREDS и содержит двоичные представления десятичных значений 0, 100, 200, ..., 900.

3.41. Написанная вами для упражнения 3.40 программа, которая выполняет преобразование из десятичного формата в двоичный, должна содержать две вложенные подпрограммы. Главная подпрограмма вызывает первую подпрограмму и передает ей два параметра через стек процессора. Первым параметром является адрес 3-байтового буфера памяти для хранения введенных десятичных цифр. Второй параметр — это адрес, по которому должно быть записано результирующее двоичное значение. Первая подпрограмма считывает с клавиатуры три символа и для выполнения преобразования вызывает вторую подпрограмму. На этот раз параметры передаются через регистры процессора. Обе подпрограммы должны сохранять содержимое используемых ими регистров в стеке процессора.

а) Напишите две указанные подпрограммы на языке ассемблера процессора 68000.

б) Приведите содержимое стека сразу после выполнения команды вызова второй подпрограммы.

3.42. Рассмотрим массив 16-разрядных чисел A(i,j), где индекс строки i принимает значения из диапазона от 0 до $n - 1$, а индекс столбца j — значения из диапазона от 0 до $m - 1$. Этот массив хранится в памяти компьютера с процессором 68000 построчно, а элементы каждой строки занимают m последовательных слов. Напишите подпрограмму для процессора 68000 для поэлементного сложения столбцов x и y с записью сумм в столбец y. Индексы x и y передаются подпрограмме в регистрах D1 и D2, параметры n и m — в регистрах D3 и D4, а адрес элемента A(0,0) — в регистре A0. Допускается использование любого режима адресации из числа указанных в табл. 3.2.

3.43. Напишите программу ARM для изменения порядка битов в регистре D2 на обратный. Например, если первоначально в регистре D2 содержится значение 1110 ... 0100, то после выполнения вашей программы в нем должно содержаться значение 0010 ... 0111. (Подсказка: используйте операции сдвига и циклического сдвига.)

3.44. Сколько байтов памяти необходимо для хранения программы, приведенной на рис. 3.32? Сколько обращений к памяти она выполняет?

3.45. Приведите результаты трассировки программы сортировки байтов, показанной на рис. 3.34, *б*, в формате, описанном в упражнении 3.27. Каким будет содержимое регистров D1, D2 и D3 и последовательности из 5 байт, расположенных по адресам LIST, LIST + 1, ..., LIST + 4, после каждого выполнения последней команды программы. Предполагается, что LIST = 1000, начальными значениями байтов являются 120, 13, 106, 45 и 67, а [LIST] = 120.

3.46. Перепишите приведенную на рис. 3.34, *б* программу сортировки байтов в виде подпрограммы, сортирующей список последовательно расположенных в памяти 32-разрядных положительных целых чисел. Вызывающая программа должны передавать подпрограмме адрес этого списка. Первое 32-разрядное число по указанному адресу определяет количество элементов списка, а за ним следуют сортируемые значения.

3.47. Рассмотрим программу сортировки байтов, приведенную на рис. 3.34, *б*. На каждом проходе по подсписку от LIST(j) до LIST(0) все элементы списка, для которых LIST(k) > LIST(j), меняются местами. В качестве альтернативы можно отследить адрес наибольшего значения в подсписке и в конце просмотра такового выполнить не более одного обмена. Перепишите программу с использованием этого подхода. Каковы его преимущества?

3.48. Предположим, что список полученных студентами оценок, показанный на рис. 2.14, хранится в памяти в виде связного списка, как на рис. 2.36. Напишите программу для процессора 68000, выполняющую ту же задачу, что и программа на рис. 2.15. Первая запись списка хранится в памяти по адресу 1000. Все элементы списка являются длинными словами.

3.49. Подпрограмма вставки записи в связный список, приведенная на рис. 3.35, не проверяет код вставляемой записи на предмет совпадения с кодами имеющихся в списке записей. А что произойдет в случае подобного совпадения? Модифицируйте подпрограмму таким образом, чтобы в указанном случае она возвращала в регистре A6 адрес записи, код которой совпадает с кодом данной записи, или 0, если вставка выполнена успешно.

3.50. Подпрограмма удаления записи из связного списка (рис. 3.36) не проверяет наличия в списке записи с указанным в регистре RIDNUM кодом. Модифицируйте ее таким образом, чтобы в случае успешного удаления она возвращала в регистре RIDNUM значение 0, и оставляла содержимое данного регистра неизменным, если запись в списке не будет найдена.

Процессоры Intel IA-32

3.51. В регистры и память компьютера ARM включена следующая информация.

Регистр EBX содержит значение 1000.

Регистр ESI содержит значение 2.

Числа 1, 2, 3, 4, 5 и 6 хранятся в последовательных словах памяти начиная с адреса 1000.

Метка LOC представляет адрес 1008.

Каким будет результат выполнения каждой из трех следующих команд при указанных начальных значениях?

```
а) MOV   EAX,10
   ADD   EAX,[EBX + ESI*4 + 8]
б) PUSH  20
   PUSH  30
   POP   EAX
   POP   EBX
   SUB   EAX,EBX
в) LEA   EAX,LOC
   MOV   EBX,[EAX]
```

3.52. Выполнение каких из перечисленных ниже команд IA-32 приведет к выводу ассемблером сообщения о синтаксической ошибке: Почему?

```
а) ADD   EAX,EAX
б) ADD   [EAX],[EBX + 4]
в) SUB   EAX,[EBX + ESI*4 + 20]
г) SUB   EAX,[EBX + ESI*10]
д) ADD   EAX,–31728542
е) MOV   20,EAX
ж) MOV   EAX,[EBP + ESP*4]
```

3.53. В результате *трассировки* программы может быть получен листинг содержимого определенных регистров и адресов памяти в заданные моменты ее выполнения. Каким будет содержимое регистров EAX, EBX и ECX каждый раз после трехкратного выполнения команды LOOP в программе, показанной на рис. 3.40, *б*. Представьте результаты в виде таблицы с именами трех регистров в заголовках столбцов. Содержимое регистров после каждого выполнения команды LOOP укажите в трех строках таблицы. Данные для программы приведены на рис. 3.42.

3.54. Напишите программу для процессора ARM, сравнивающую значения соответствующих байтов из двух списков байтов и помещающую большее из значений в третий список. Два исходных списка начинаются по адресам X и Y, а результирующий список — по адресу LARGER. Длина списка хранится в памяти по адресу N.

3.55. Напишите программу для процессора IA-32, которая выполняла бы ряд задач, связанных с обработкой символов. Строка из n символов хранится в памяти в виде последовательности байтов начиная с адреса STRING. Еще одна, более короткая строка из m символов хранится в памяти по адресу SUBSTRING. Программа должна просмотреть строку, начинающуюся по адресу STRING, чтобы определить, содержится ли в ней непрерывная подстрока, идентичная строке по адресу SUBSTRING. Параметры n и m ($n > m$)

хранятся в памяти по адресам N и M соответственно. Если подстрока найдена, в регистр EAX должен быть записан ее адрес. В противном случае регистр EAX должен быть очищен (установлен в 0). Многократные вхождения строки программа выявлять не должна — нужен только адрес первого вхождения подстроки в строку.

3.56. Напишите программу для процессора IA-32, генерирующую первые n чисел последовательности Фибоначчи. В этой последовательности первыми двумя числами являются 0 и 1, а каждое следующее число генерируется путем сложения двух предшествующих чисел. Например, для $n = 8$ последовательность Фибоначчи такова:

0, 1, 1, 3, 3, 5, 8, 13

Ваша программа должна записать эти числа в последовательные слова памяти начиная с адреса MEMLOC. Значение n хранится по адресу N.

3.57. Напишите программу для процессора IA-32, выполняющую преобразование слова текста, набранного в нижнем регистре, в слово в верхнем регистре. Слово состоит из символов ASCII, хранящихся в памяти в последовательных байтах начиная с адреса WORD, и заканчивается символом пробела. (Сведения об ASCII-кодах представлены в приложении Д.)

3.58. Список полученных студентами оценок (рис. 2.14) изменился, и теперь для каждого студента в нем содержится j оценок. Каждый элемент в списке является 16-разрядным словом, так что значение счетчика каждый раз должно увеличиваться на 2. Количество студентов равно n. Напишите программу для процессора IA-32, вычисляющую сумму оценок по каждому из тестов и записывающую эти суммы в двойные слова памяти по адресам SUM, SUM + 4, SUM + 8, ... Количество тестов j больше количества регистров процессора, так что для суммирования оценок нельзя применить программу, аналогичную приведенной на рис. 2.15. Воспользуйтесь вложенным циклом, как предлагалось в разделе 2.5.3. Во внутреннем цикле должна накапливаться сумма по одному тесту, а во внешнем — выполняться проход по всем тестам. Количество тестов j хранится в памяти по адресу J, который меньше адреса N.

3.59. Напишите программу для процессора IA-32, которая изменяла бы порядок битов в регистре EAX на обратный. Например, если первоначально в регистре EAX хранится значение 1110...0100, после выполнения вашей программы в нем должно содержаться значение 0010...0111. (Подсказка: используйте операции сдвига и циклического сдвига.)

3.60. Воспользовавшись очередью, описанной в упражнении 2.18, создайте подпрограммы APPEND и REMOVE для процессора IA-32, которые выполняли бы пересылку данных между регистрами процессора и очередью. Будьте внимательны при анализе и обновлении состояния очереди и указателей.

3.61. Напишите подпрограмму для процессора IA-32, которая считывала бы с клавиатуры n символов, помещала их по мере ввода в пользовательский стек и отображала на дисплее. В качестве указателя стека используйте регистр EBX. Информация о количестве символов содержится в памяти по адресу N.

3.62. Предположим, что среднее время выборки и выполнения команды программы, приведенной на рис. 3.44, составляет 10 нс. Если символы вводятся с клавиатуры со скоростью 10 символов в секунду, сколько раз для каждого из них будет выполнена команда JNC READ? Время, уходящее на вывод символа на экран, много меньше времени, проходящего между вводом последовательных символов с клавиатуры.

3.63. В программе для процессора IA-32 (рис. 3.44) чтение и вывод символов выполняется без вызова подпрограмм. Перепишите эту программу таким образом, чтобы она состояла из главной программы, вызывающей подпрограмму GETCHAR для ввода одного символа и подпрограмму PUTCHAR для вывода одного символа. Адреса регистров INSTATUS и DATAIN передаются подпрограмме GETCHAR в регистрах EBX и EDX, а считанный с клавиатуры символ главная подпрограмма получает через регистр AL. Адреса регистров OUTSTATUS, DATAOUT и подлежащий выводу символ передаются подпрограмме PUTCHAR в регистрах ESI, EDI и AL соответственно. Любые другие используемые в подпрограммах регистры должны сохраняться и восстанавливаться с применением стека процессора, указатель на который находится в регистре ESP. Запись символов в память и проверка на наличие символа конца строки CR должна выполняться главной программой.

3.64. Повторите упражнение 3.63 при условии, что параметры будут передаваться через стек процессора.

3.65. Напишите программу для процессора IA-32, выполняющую ввод с клавиатуры трех цифр. Каждая цифра представлена ASCII-кодом. Вместе эти цифры составляют десятичное целое число из диапазона от 0 до 999. Первой вводится старшая цифра. Преобразуйте это число в двоичное представление. Чтобы упростить процесс преобразования, в память помещены две таблицы слов, содержащие по десять элементов. Первая из них начинается по адресу TENS. В ней содержатся двоичные представления десятичных значений 0, 10, 20, ..., 90. Вторая таблица начинается по адресу HUNDREDS и содержит двоичные представления десятичных значений 0, 100, 200, ..., 900.

3.66. Написанная для упражнения 3.65 программа, выполняющая преобразование из десятичного формата в двоичный, должна содержать две вложенные подпрограммы. Главная подпрограмма вызывает первую подпрограмму и передает ей два параметра через стек процессора. Первым параметром является адрес 3-байтового буфера памяти для хранения введенных десятичных цифр. Второй параметр — это адрес, по которому должно быть записано результирующее двоичное значение. Первая подпрограмма считывает с клавиатуры три символа и вызывает вторую подпрограмму для выполнения преобразования. В этом случае параметры передаются через регистры процессора. Обе подпрограммы должны сохранять содержимое используемых ими регистров в стеке процессора.

а) Напишите эти подпрограммы на языке ассемблера процессора IA-32.

б) Каковым будет содержимое стека сразу после выполнения команды вызова второй подпрограммы.

3.67. Рассмотрим массив 16-разрядных чисел A(i,j), где индекс строки i принимает значения из диапазона от 0 до n − 1, а индекс столбца j — из диапазона от 0 до m − 1. Этот массив хранится в памяти компьютера с процессором IA-32 построчно, а элементы каждой строки занимают m последовательных двойных слов. Напишите подпрограмму для поэлементного сложения столбцов x и y с записью сумм в столбец y. Индексы x и y передаются подпрограмме в регистрах D1 и D2, параметры n и m — в регистрах ESI и EDI, а адрес элемента A(0,0) — в регистре EDX. Допускается использование любых режимов адресации из перечисленных в табл. 3.3.

3.68. Приведите результаты трассировки представленной на рис. 3.50, *б* программы сортировки байтов в формате, описанном в упражнении 3.53. Покажите содержимое регистров EDI, ECX и DL и последовательности из пяти байтов, расположенных по адресам LIST, LIST + 1, ..., LIST + 4, после каждого выполнения последней команды программы. Предполагается, что LIST = 1000, начальными значениями байтов являются 120, 13, 106, 45 и 67, а [LIST] = 120.

3.69. Перепишите приведенную на рис. 3.50, *б* программу сортировки байтов в виде подпрограммы, сортирующей список последовательно расположенных в памяти 32-разрядных положительных целых чисел. Вызывающая программа должны передавать подпрограмме адрес этого списка. Первое 32-разрядное число по указанному адресу определяет количество элементов списка, а за ним следуют сортируемые значения.

3.70. Рассмотрим программу сортировки байтов, приведенную на рис. 3.50, *б*. На каждом проходе по подсписку от LIST(j) до LIST(0) все элементы списка, для которых LIST(k) > LIST(j), меняются местами. В качестве альтернативы можно отслеживать адрес наибольшего значения в подсписке и в конце просмотра подсписка выполнять не более одного обмена. Перепишите программу с использованием предлагаемого подхода. Каковы его преимущества?

3.71. Предположим, что список полученных студентами оценок, представленный на рис. 2.14, хранится в памяти в виде связного списка, как показано на рис. 2.36. Напишите программу для процессора IA-32, выполняющую ту же задачу, что и программа на рис. 2.15. Первая запись списка хранится в памяти по адресу 1000.

3.72. Подпрограмма вставки записи в связный список, приведенная на рис. 3.51, не проверяет код вставляемой записи на предмет совпадения с кодами имеющихся в списке записей. Что произойдет в случае такого совпадения? Модифицируйте подпрограмму так, чтобы она возвращала в регистре EDX адрес записи, код которой совпадает с кодом вставляемой записи, или 0, если вставка выполнена успешно.

3.73. Подпрограмма удаления записи из связного списка (рис. 3.52), не проверяет наличия в списке записи с указанным в регистре RIDNUM кодом. Модифицируйте ее таким образом, чтобы в случае успешного удаления она возвращала в регистре RIDNUM значение 0 и оставляла содержимое данного регистра неизменным, если запись в списке не найдена.

Глава 4

Ввод-вывод

- Выполнение программного ввода-вывода с использованием пулинга
- Прерывания и необходимое для их поддержки аппаратное и программное обеспечение
- Прямой доступ к памяти и механизмы ввода-вывода для высокоскоростных устройств
- Пересылка данных через синхронные и асинхронные шины
- Разработка схем интерфейса ввода-вывода
- Коммерческие стандарты шинной архитектуры, в частности PCI, SCSI и USB

Одной из важнейших функций компьютера является поддержка устройств ввода-вывода. Благодаря этому оператор, к примеру, может использовать клавиатуру и дисплей для работы с текстом и графикой. Компьютеры применяются для взаимодействия с другими компьютерами через Интернет, а следовательно, предоставляют доступ к информации, находящейся в различных частях земного шара. Роль компьютеров в решении других задач может быть менее заметной, но не менее важной. Компьютеры используются дома, в учебных аудиториях, на производстве, в различных транспортных, банковских и коммерческих системах — разнообразие сфер их применения просто впечатляет. Данные могут поступать в компьютер от самых разных приборов: сенсорных устройств, видеокамер, в том числе цифровых, микрофонов или даже с пультов пожарной сигнализации. Выходными же данными могут служить направляемый в колонки звуковой сигнал или, скажем, закодированная цифровая команда, изменяющая скорость мотора, открывающая вентиль или заставляющая робота выполнить определенное движение. Короче говоря, компьютер должен обладать способностью обмениваться информацией с широким диапазоном устройств в различных окружениях.

В настоящей главе мы подробно рассмотрим разнообразные способы выполнения операций ввода-вывода. Прежде всего данный вопрос будет проанализирован с точки зрения программиста. Затем речь пойдет о некоторых аппаратных деталях, связанных с шинами и интерфейсами ввода-вывода. Напоследок вы познакомитесь с несколькими наиболее распространенными стандартами шинной архитектуры.

4.1. Доступ к устройствам ввода-вывода

Простейшая схема подключения устройств ввода-вывода к компьютеру заключается в использовании общей шины (рис. 4.1). Все устройства, подключенные к шине, могут обмениваться между собой информацией. Обычно шина состоит из трех наборов линий, предназначенных для передачи адресов, данных и управляющих сигналов. Каждому устройству ввода-вывода присваивается уникальный набор адресов. Когда процессор помещает на адресные линии конкретный адрес, распознавшее этот адрес устройство отвечает на команду, помещенную на управляющие линии. Процессор запрашивает либо операцию чтения, либо операцию записи, и запрошенные данные пересылаются по линиям данных. Как упоминалось в разделе 2.7, организация системы ввода-вывода, при которой устройства ввода-вывода и память разделяют одно адресное пространство, называется *вводом-выводом с отображением в память*.

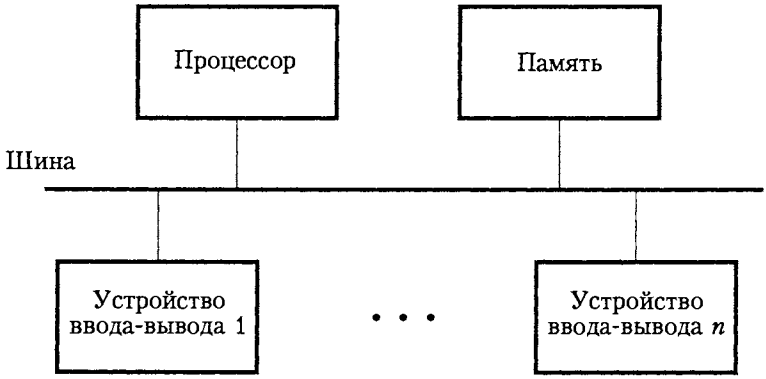


Рис. 4.1. Архитектура системы с общей шиной

При использовании ввода-вывода с отображением в память любые машинные команды, выполняющие обращение к памяти, могут быть задействованы и для обмена данными с устройствами ввода-вывода. Предположим, DATAIN — это адрес входного буфера, связанного с клавиатурой. Тогда следующая команда считывает данные из DATAIN и помещает их в регистр процессора R0:

```
Move  DATAIN,R0
```

Аналогичным образом команда

```
Move  R0,DATAOUT
```

пересылает содержимое регистра R0 по адресу DATAOUT, который может соответствовать выходному буферу дисплея или принтера.

Технология ввода-вывода с отображением в память применяется в большинстве компьютерных систем. Некоторые процессоры для выполнения операций ввода-вывода поддерживают специальные команды In и Out. Например, процессоры семейства Intel, описанного в главе 3, имеют специальные команды ввода-вывода

и отдельное 16-разрядное адресное пространство для устройств ввода-вывода. Создавая компьютерную систему на основе такого процессора, конструктор может либо соединить все устройства ввода-вывода так, чтобы они использовали специальное адресное пространство ввода-вывода, либо подключить их как часть адресного пространства памяти. Второй подход позволяет упростить программное обеспечение, поэтому используется он значительно чаще. Одним из преимуществ наличия отдельного адресного пространства ввода-вывода является то, что соответствующие устройства могут использовать меньшее количество адресных линий. Но имейте в виду, что наличие отдельного адресного пространства ввода-вывода еще не означает, что адресные линии ввода-вывода физически отделены от адресных линий памяти. На тот факт, что запрошенная операция чтения или записи относится к системе ввода-вывода, может указывать передаваемый по шине специальный сигнал. Получив такой сигнал, память игнорирует запрошенную операцию, а устройства ввода-вывода анализируют младшие разряды переданного по шине адреса, чтобы узнать, кому из них направлен запрос.

Аппаратные элементы, необходимые для присоединения устройств ввода-вывода к шине, представлены на рис. 4.2. Когда адрес устройства появляется на адресных линиях, устройство распознает его с помощью дешифратора (декодера) адреса. Данные, которыми устройство обменивается с процессором, хранятся в регистрах данных. Регистр состояния содержит информацию, относящуюся к функционированию устройства ввода-вывода. Регистры данных и состояния соединяются шиной данных, и им присваиваются уникальные адреса. Дешифратор адреса, регистры данных и состояния, управляющие схемы, необходимые для координирования операций ввода-вывода, составляют *схему сопряжения*, или *интерфейс*, устройства.

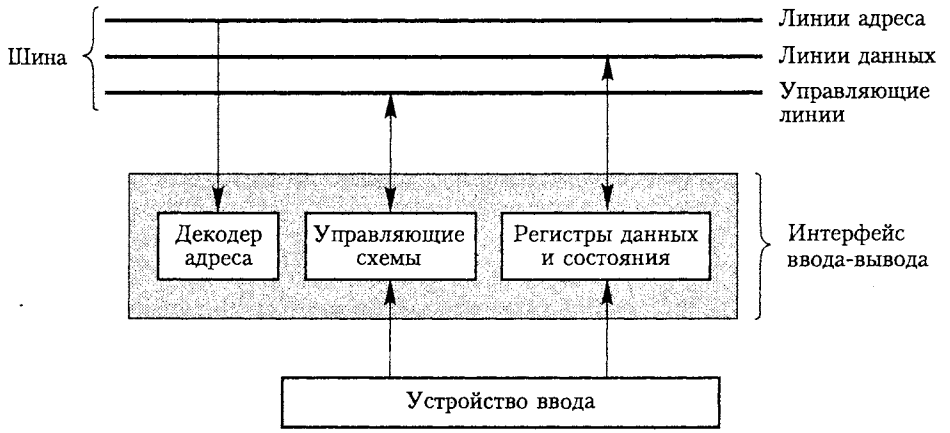


Рис. 4.2. Интерфейс ввода-вывода для устройства ввода

Скорость работы устройств ввода-вывода значительно отличается от скорости работы процессора. Когда оператор вводит с клавиатуры символы, между последовательными нажатиями клавиш процессор может выполнять миллионы команд.

Команда, считывающая символ с клавиатуры, должна выполняться только тогда, когда таковой находится во входном буфере интерфейса клавиатуры. Кроме того, необходимо гарантировать, что один и тот же символ не будет прочитан из этого буфера дважды.

Основные идеи, положенные в основу операций ввода-вывода, были изложены в разделе 2.7. В случае входного устройства, подобного клавиатуре, в схему сопряжения в виде одного из разрядов регистра состояния включается флаг состояния SIN. Он устанавливается в 1, если символ вводится с клавиатуры, и сбрасывается в 0, если символ считывается процессором. Таким образом, проверяя значения флага SIN, программное обеспечение гарантирует корректность операции чтения данных. Для этого обычно организуется программный цикл, считывающий регистр состояния и проверяющий состояние флага SIN. Обнаружив, что флаг установлен в 1, программа считывает значение из регистра входных данных. Аналогичным образом может осуществляться управление операциями вывода, но в этом случае применяется флаг состояния SOUT.

Пример 4.1

Для того чтобы лучше усвоить базовые концепции ввода-вывода, мы рассмотрим простой пример с участием клавиатуры и дисплея. Для операций пересылки данных используются четыре регистра (рис. 4.3). В регистре STATUS содержатся два управляющих флага, SIN и SOUT, хранящие соответственно информацию о состоянии клавиатуры и дисплея. Еще два флага из этого регистра, KIRQ и DIRQ, используются при обработке прерываний. О флагах KEN и DEN, относящихся к регистру CONTROL, будет рассказано в разделе 4.2. Вводимые с клавиатуры данные помещаются в регистр DATAIN, а данные, отправляемые на дисплей, — в регистр DATAOUT.

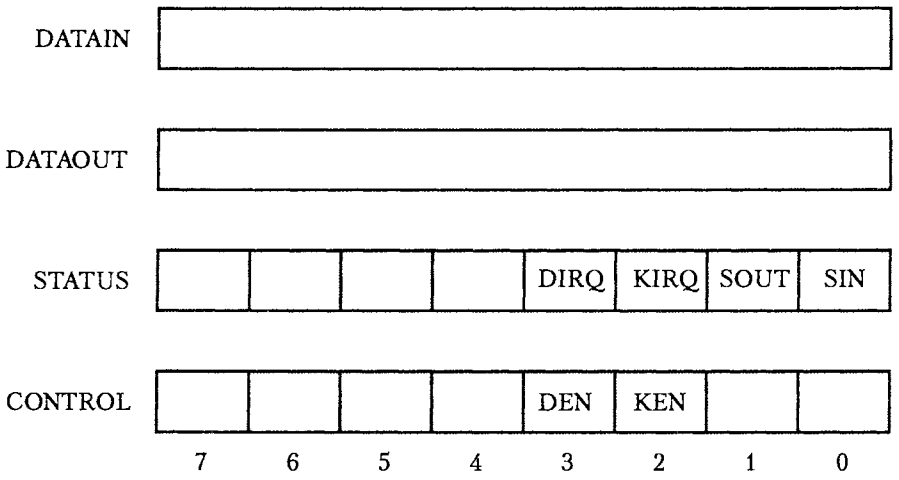


Рис. 4.3. Регистры интерфейса клавиатуры и дисплея

	Move	#LINE,R0	Инициализация указателя на память
WAITK	TestBit	#0,STATUS	Проверка флага SIN
	Branch=0	WAITK	Ожидание ввода символа
	Move	DATAIN,R1	Чтение символа
WAITD	TestBit	#1,STATUS	Проверка флага SOUT
	Branch=0	WAITD	Ожидание момента готовности дисплея
	Move	R1,DATAOUT	Отправка символа на дисплей
	Move	R1,(R0)+	Сохранение символа в памяти и увеличение значения указателя
	Compare	#$0D,R1	Проверка того, введен ли символ возврата каретки
	Branch≠0	WAITK	Если символ возврата каретки не введен, считывается следующий символ
	Move	#$0A,DATAOUT	Если символ возврата каретки введен, на дисплей отправляется символ перевода строки
	Call	PROCESS	Вызов подпрограммы для обработки введенной строки

Рис. 4.4. Программа, которая считывает с клавиатуры одну строку, сохраняет ее в буфере памяти, а затем отображает на дисплее

Программа, приведенная на рис. 4.4, похожа на подпрограмму, представленную на рис. 2.20. Она считывает с клавиатуры строку символов и сохраняет ее в памяти, в буфере, начинающемся с адреса LINE. Затем она вызывает подпрограмму PROCESS, выполняющую обработку введенной строки. Каждый вводимый символ отображается на дисплее. Регистр R0 используется как указатель на буфер в памяти. Его содержимое обновляется с применением автоинкрементного режима адресации, благодаря чему последовательно вводимые символы сохраняются в памяти по последовательным адресам.

После ввода каждого символа подпрограмма проверяет, не является ли он символом возврата каретки (шестнадцатеричным ASCII-кодом 0D). Если он таковым является, на дисплей отправляется символ перевода строки (шестнадцатеричный ASCII-код 0A), перемещающий курсор на одну строку вниз. Затем вызывается подпрограмма PROCESS. Если введен любой другой символ, программа переходит на начало цикла и ждет ввода следующего символа.

Данный пример иллюстрирует принцип технологии программно-управляемого ввода-вывода, при котором процессор постоянно проверяет флаг состояния, чтобы синхронизировать свою работу с работой внешнего устройства. Мы говорим, что процессор *опрашивает* устройство. Существует еще два распространенных механизма реализации ввода-вывода: прерывания и прямой доступ к памяти. Если для управления вводом-выводом используются прерывания, синхронизация достигается за счет того, что устройство ввода-вывода само сообщает о своей готовности, отправляя через шину специальный сигнал. Прямой доступ к памяти является характерным для высокоскоростных устройств ввода-вывода. Эта технология позволяет интерфейсным схемам устройства самостоятельно обмениваться

данными с памятью, без постоянного участия процессора. Эти два механизма мы обсудим в следующих разделах, после чего поговорим о необходимых для их реализации аппаратных компонентах — шине процессора и интерфейсах устройств ввода-вывода.

4.2. Прерывания

В примере на рис. 4.4 программа выполняет цикл ожидания, постоянно проверяя состояние устройства. В течение этого времени процессор не делает никакой полезной работы. Однако во многих случаях в течение времени ожидания готовности устройства ввода-вывода процессор мог бы выполнять другие задачи. Для этого нужно так организовать совместную работу процессора и внешнего устройства, чтобы это устройство само оповещало процессор о своей готовности к пересылке данных. Такое оповещение выполняется с помощью специального сигнала, называемого *прерыванием*. Для прерываний обычно выделяется как минимум одна управляющая линия шины, называемая *линией запроса прерывания*. Поскольку при использовании прерываний процессору не нужно постоянно проверять состояние внешних устройств, он может использовать периоды ожидания для выполнения других полезных функций. Благодаря этому процессор никогда не простаивает.

Пример 4.2

Рассмотрим задачу, для выполнения которой нужно произвести некоторые вычисления и напечатать результаты на принтере, затем произвести другие вычисления и снова распечатать результаты — и так несколько раз. Предположим, что выполняющая эту задачу программа состоит из двух подпрограмм, COMPUTE и PRINT. Подпрограмма COMPUTE генерирует n выходных строк, которые должны быть напечатаны подпрограммой PRINT.

Для реализации указанной задачи программа может многократно выполнять подпрограмму COMPUTE, а затем подпрограмму PRINT. Принтер за один раз принимает только одну строку текста. Поэтому подпрограмма PRINT должна отослать одну строку текста, подождать, пока он будет напечатан, отослать следующую строку и т. д. Недостатком этого подхода является то обстоятельство, что процессор тратит немало времени в ожидании готовности принтера. Общую скорость выполнения программы можно значительно увеличить путем чередования процессов печати и вычисления, то есть за счет выполнения во время печати подпрограммы COMPUTE. Делается это следующим образом. Сначала выполняется подпрограмма COMPUTE, которая генерирует n первых строк текста, затем — подпрограмма PRINT, отправляющая первую строку на принтер. Однако вместо того чтобы дожидаться окончания процесса печати строки, подпрограмма PRINT может временно приостановить свое выполнение и передать управление программе COMPUTE. Когда принтер будет готов к печати следующей строки, он оповестит об этом процессор, выдав сигнал запроса прерывания. В ответ процессор прервет выполнение подпрограммы COMPUTE и передаст управление подпрограмме

PRINT. Подпрограмма PRINT отправит на принтер вторую строку и снова приостановит свою работу. Прерванная программа COMPUTE продолжит работу с точки прерывания. Этот процесс может продолжаться до тех пор, пока не будут напечатаны все *n* строк и подпрограмма PRINT не закончит свою работу.

Когда следующие *n* строк будут готовы для печати, выполнение подпрограммы PRINT начнется сначала. И если на формирование подпрограммой COMPUTE *n* строк текста понадобится больше времени, чем на их печать, процессор все время будет занят полезными вычислениями.

Этот пример иллюстрирует концепцию прерываний. Программа, выполняемая в ответ на запрос прерывания, называется *программой обработки прерываний*. В нашем случае роль таковой выполняет подпрограмма PRINT. Прерывания очень похожи на вызовы подпрограмм. Предположим, что запрос прерывания поступает во время выполнения команды *i* (рис. 4.5). Процессор завершает выполнение этой команды, а затем загружает в счетчик команд адрес первой команды программы обработки прерываний. Для упрощения задачи предположим, что этот адрес аппаратно закреплен в процессоре. После выполнения программы обработки прерываний процессор должен вернуться к команде *i* + 1. Для этого перед вызовом программы обработки прерывания содержимое регистра PC должно быть временно сохранено в памяти. Команда возврата из прерывания, расположенная в конце программы обработки прерывания, загрузит этот сохраненный адрес в регистр PC, в результате чего выполнение будет продолжено с команды *i* + 1. Во многих компьютерах адрес возврата сохраняется в стеке процессора. Но он может быть сохранен и в другом месте, например, в специально для него предназначенном регистре.

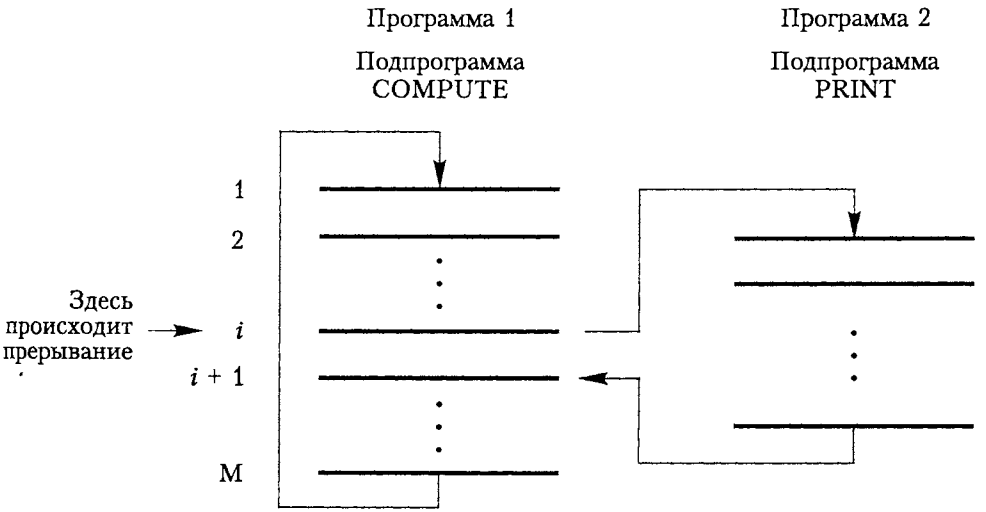


Рис. 4.5. Передача управления через прерывания

Обрабатывая прерывание, процессор должен проинформировать устройство о том, что его запрос распознан, после чего данное устройство сможет снять сигнал запроса прерывания. Для этого по шине может быть передан специальный управляющий сигнал. О сигнале подтверждения прерывания, используемом в некоторых схемах обработки прерываний, рассказывается далее в этой главе. Распространенной альтернативой такому подходу является пересылка данных между процессором и интерфейсом ввода-вывода. В программе обработки прерывания выполняется команда, которая изменяет значение в регистре состояния или регистре данных в интерфейсе устройства и тем самым явно информирует устройство о том, что запрос прерывания получен процессором.

Получается, что программа обработки прерывания очень похожа на обычную подпрограмму. Однако между ними имеются очень важные различия. Подпрограмма выполняет функцию, необходимую той программе, из которой она вызвана, тогда как программа обработки прерываний может не иметь ничего общего с выполняемой программой — эти две программы обычно даже принадлежат разным пользователям. Таким образом, перед вызовом программы обработки прерывания необходимо сохранить всю информацию, которая может быть изменена в ходе ее выполнения. Перед выходом из программы обработки прерывания эта информация должна быть восстановлена. После этого исходная программа может продолжить свое выполнение так, словно оно и не прерывалось (конечно, если не считать времени задержки). К числу сохраняемой и восстанавливаемой информации обычно относятся значения флагов условий и содержимое всех тех регистров, которые используются и прерванной программой, и программой обработки прерывания.

Задача сохранения и восстановления информации может автоматически выполняться процессором или же командами программы. Большинство современных процессоров сохраняют только минимальное количество информации, необходимое для обеспечения целостности программ. Дело в том, что процесс сохранения и восстановления регистров включает обмен данными с памятью, увеличивающий время задержки между получением запроса прерывания и вызовом программы его обработки. Такого рода задержка называется *задержкой обработки прерывания*. Для некоторых приложений слишком большая задержка обработки прерываний неприемлема. Поэтому-то количество сохраняемой и восстанавливаемой процессором информации и должно быть сведено к минимуму. Как правило, процессор сохраняет только содержимое счетчика команд и регистра состояния процессора. Любая дополнительная информация должна сохраняться программным путем в начале работы программы обработки прерывания и восстанавливаться перед ее завершением.

Некоторые ранние процессоры, и в первую очередь те, в которых было небольшое количество регистров, при получении запроса прерывания автоматически сохраняли содержимое всех своих регистров (без программного участия). Процедура восстановления регистров была включена в реализацию команды возврата из процедуры обработки прерывания. Некоторые процессоры поддерживают два типа прерываний: одни вызывают автоматическое сохранение регистров, а другие — нет. Конкретное устройство ввода-вывода может использовать любой из типов прерываний в зависимости от необходимого времени ответа. Другой интересный подход

к решению этого вопроса заключается в использовании двух дублирующих друг друга наборов регистров процессора. Это позволяет программе обработки прерывания использовать второй набор регистров, поэтому сохранять и восстанавливать их не требуется.

Прерывания — это нечто большее, нежели механизм координирования операций ввода-вывода. В общем случае прерывания обеспечивают передачу управления от одной программы к другой, инициируемую внешним по отношению к компьютеру событием. После выполнения программы обработки прерывания работа прерванной программы возобновляется. Концепция прерываний применяется в операционных системах и во многих управляющих приложениях, когда выполнение определенных подпрограмм должно точно синхронизироваться с внешними событиями. Программы такого типа называются *приложениями реального времени.*

4.2.1. Аппаратное обеспечение для поддержки прерываний

Как уже было сказано, устройства ввода-вывода запрашивают прерывания путем активизации линии шины, называемой линией запроса прерывания. В большинстве компьютеров прерывания могут запрашиваться несколькими устройствами ввода-вывода. Но и в этом случае для обслуживания прерываний может использоваться одна линия (рис. 4.6). Каждое из устройств подсоединяется к этой линии с помощью ключа, соединенного с «землей». Для того чтобы запросить прерывание, устройство замыкает этот ключ. Таким образом, если ни один из сигналов запроса прерывания от $INTR_1$ до $INTR_n$ не активен, то есть если все ключи открыты, напряжение на линии запроса прерывания равно V_{dd}. Это неактивное состояние линии. Когда устройство запрашивает прерывание, замыкая свой ключ, напряжение на линии падает до 0, в результате чего процессор получает сигнал запроса прерывания INTR, равный 1. Поскольку замыкание одного или нескольких ключей приводит к падению напряжения на линии до 0, значение INTR является логической суммой (ИЛИ) запросов отдельных устройств:

$$INTR = INTR_1 + ... + INTR_n$$

Для обозначения сигнала запроса прерывания часто используют форму дополнения $\overline{INTR}$, поскольку сигнал активен, когда напряжение на линии равно 0.

В электронной реализации схемы, показанной на рис. 4.6, для управления линией $\overline{INTR}$ используются специальные вентили, называемые *вентилями с открытым коллектором* (для двухполюсных схем) или вентилями *с открытым стоком* (для схем МОП). Такие вентили эквивалентны ключу, соединяющему линию с «землей»: когда ключ открыт, значением на выходе вентиля является 0, а когда закрыт — 1. Уровень напряжения, а также логическое состояние выхода вентиля в соответствии с приведенным выше равенством зависят от сигналов, поданных на все соединенные с шиной вентили. Резистор R называется *повышающим резистором,* поскольку при открытых ключах он позволяет поднять напряжение на линии до уровня, соответствующего состоянию высокого напряжения.

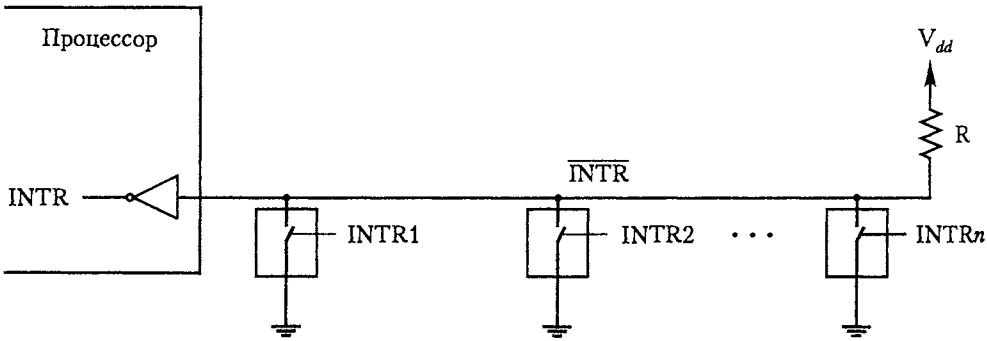


Рис. 4.6. Архитектура шины с открытым стоком, используемая при реализации типичной линии запроса прерывания

4.2.2. Запрет и разрешение прерываний

Компьютер должен предоставлять программисту полный контроль над событиями, происходящими во время выполнения программы. В ответ на поступивший от внешнего устройства запрос на прерывание процессор приостанавливает выполнение одной программы и начинает выполнение другой. Поскольку запросы на прерывание могут поступать в любой момент, они наверняка изменят последовательность событий, установленную программистом. Поэтому выполнение программ обработки прерываний должно тщательно контролироваться. Одной из основных возможностей, которой должен обладать любой компьютер, является возможность запрещать и разрешать прерывания по мере необходимости. Давайте поговорим об этом более детально.

Существует множество ситуаций, когда процессор должен игнорировать запросы прерываний. Например, в программе, выполняющей вычисления и печать (рис. 4.5), прерывания от принтера должны обрабатываться только в том случае, если имеются подготовленные к печати выходные строки. После печати n строк прерывания должны быть запрещены до тех пор, пока не будет готов следующий набор строк. В другой ситуации, возможно, понадобится дать гарантию того, что заданная последовательность команд будет выполнена до конца без прерываний, поскольку программа обработки прерываний может изменить используемые этими командами данные. Следовательно, в распоряжении программиста должны быть средства как для разрешения прерываний, так и для их запрета. Проще всего для этих целей иметь специальные машинные команды.

Рассмотрим простой пример обработки запроса на прерывание, поступающего от одного устройства. Когда устройство активизирует сигнал запроса прерывания, оно поддерживает этот сигнал активным до тех пор, пока не узнает, что процессор принял запрос. Это означает, что сигнал запроса прерывания будет активен еще какое-то время спустя после вызова программы обработки прерывания – до тех пор, пока не будет выполнена команда обращения к данному устройству. Важно, чтобы этот активный сигнал не привел к следующему прерыванию и таким образом не заставил систему войти в бесконечный цикл, из которого она не сможет выйти. Существует несколько механизмов решения этой проблемы. Три

из них мы рассмотрим сейчас, а остальные, предусматривающие обработку прерываний от более чем одного устройства, будут описаны ниже.

Первый механизм заключается в игнорировании схемой процессора сигнала на линии запроса прерывания до окончания выполнения первой команды в программе обработки прерывания. При этом первая команда программы обработки должна запретить прерывания до окончания действия данной программы. Как правило, команда, разрешающая прерывания, является последней командой программы его обработки (она предшествует команде возврата из прерывания). При этом процессор должен гарантировать, что выполнение команды возврата из прерывания будет завершено до того, как станут возможными следующие прерывания.

Второй механизм в большей мере подходит для простого процессора с единственной линией запроса прерывания. Он заключается в том, что процессор сам запрещает прерывания перед началом выполнения программы обработки прерываний и разрешает таковые по ее завершении. После сохранения в стеке регистра PC и регистра состояния процессора PS процессор выполняет действия, эквивалентные команде запрета прерываний. Очень часто для запрета и разрешения прерываний используется один разряд в регистре PS, называемый *флагом разрешения прерываний* (interrupt-enable). Если прерывания разрешены, этот разряд содержит 1, а если запрещены — 0. После сохранения в стеке регистра PS, в котором разряд разрешения прерываний установлен в 1, процессор очищает этот разряд в своем регистре PS, запрещая тем самым дальнейшие прерывания. При выполнении команды возврата из прерывания содержимое регистра PS восстанавливается из стека и флаг разрешения прерываний снова становится равным 1. Это значит, что прерывания разрешены.

Третий механизм предполагает, что у процессора имеется специальная линия запроса прерываний и что схема управления прерываниями отвечает только на передний фронт сигнала. Эта линия называется управляемой фронтом сигнала. При такой схеме работы процессор получает только один запрос прерывания, независимо от того, как долго линия остается активной. Это значит, что повторяющихся прерываний быть не может и нет необходимости явно отключать запросы прерывания на данной линии.

Прежде чем мы перейдем к более сложным аспектам прерываний, давайте еще раз коротко определим последовательность событий, происходящих в ходе обработки запроса прерывания от одного устройства. Если предположить, что изначально прерывания разрешены, эта последовательность будет следующей.

1. Устройство генерирует запрос прерывания.
2. Процессор прерывает текущую выполняемую программу.
3. Последующие прерывания запрещаются, для чего изменяются управляющие биты в регистре PS (за исключением схем, в которых линия запроса прерывания управляется фронтом сигнала).
4. Устройство информируется о том, что его запрос распознан, и в ответ сбрасывает сигнал запроса на прерывание.
5. Запрошенное прерыванием действие выполняется программой обработки прерывания.
6. Прерывания разрешаются, выполнение программы возобновляется.

4.2.3. Обслуживание нескольких устройств

Рассмотрим случай, когда с процессором соединено несколько устройств, способных инициировать прерывания. Поскольку эти устройства функционально независимы, они генерируют прерывания без какой-либо определенной последовательности. Например, устройство X может запросить прерывание во время обслуживания прерывания от устройства Y или несколько устройств могут запросить прерывания одновременно. В связи с этим возникает ряд вопросов:

1. Как процессор распознает устройство, запросившее прерывание?
2. Если разным устройствам требуются различные программы обработки прерываний, как процессор в каждом случае будет получать начальный адрес соответствующей программы?
3. Можно ли устройствам прерывать процессор, пока обслуживается другое прерывание?
4. Как должны обрабатываться два или несколько одновременно поступивших запросов на прерывания?

Существуют разные способы решения этих вопросов, и при выборе компьютера для определенных задач обычно учитывают, какая стратегия в нем используется.

Итак, когда процессор получает запрос прерывания по общей линии (рис. 4.6), ему требуется дополнительная информация, для того чтобы определить, какое из устройств активизировало эту линию. Далее, если два устройства активизировали линию одновременно, нужно выбрать одно из них для обслуживания. Когда первое устройство будет обслужено, наступит очередь второго.

Информация, необходимая для идентификации устройства, запросившего данное прерывание, имеется в его регистре состояния. Когда устройство генерирует запрос прерывания, оно устанавливает в 1 один из разрядов в регистре состояния, называемый разрядом IRQ. Например, сигналы запросов на прерывания от клавиатуры и дисплея устанавливают в 1 биты KIRQ и DIRQ (см. рис. 4.3). Простейший способ определения устройства, запросившего прерывание, заключается в опросе всех присоединенных к шине устройств ввода-вывода. Сначала обслуживается устройство, у которого разряд IRQ был установлен первым. Для обработки запроса вызывается соответствующая программа.

Реализовать описанную схему достаточно легко. Ее главным недостатком является время, уходящее на проверку IRQ-разрядов тех устройств, которые не запрашивали прерывание. В качестве альтернативы могут использоваться векторные прерывания, о которых рассказывается в следующем разделе.

Векторные прерывания

Для сокращения времени, уходящего на опрос устройств, можно реализовать схему, обратную описанной выше: устройство само будет идентифицировать себя для процессора. Это позволит процессору начинать выполнение программы обработки прерывания сразу же после поступления запроса. Термин *векторные прерывания* относится ко всех схемам обработки прерываний, основанным на таком подходе.

Устройство, запросившее прерывание, может идентифицировать себя с помощью специального кода, пересылаемого процессору по шине, что позволяет ему, процессору, идентифицировать отдельные устройства даже в том случае, если они используют одну линию запроса прерывания. Этот код способен определять начальный адрес программы обработки прерывания, предназначенной для данного устройства. Длина адреса обычно составляет от 4 до 8 разрядов. Оставшаяся часть адреса формируется процессором на основе данных, хранящихся в специально выделенной области памяти с адресами программ обработки прерываний.

При этом предполагается, что программа обработки прерываний от конкретного устройства должна всегда располагаться по одному и тому же адресу. Для обеспечения большей ее гибкости программист может поместить по этому адресу команду, выполняющую переход к соответствующей подпрограмме. Во многих компьютерах такой переход автоматически выполняется механизмом обработки прерываний. То место в памяти, на которое указывает вызвавшее прерывание устройство, используется для хранения начального адреса программы обработки прерывания. Процессор считывает этот адрес, называемый *вектором прерывания*, и загружает его в регистр PC. Кроме адреса вектор прерывания может содержать новое значение регистра состояния процессора.

В большинстве компьютеров устройства ввода-вывода направляют код вектора прерывания по шине данных с использованием управляющих сигналов шины, и это является гарантией того, что устройства не будут мешать друг другу. Когда устройство направляет процессору запрос прерывания, тот, возможно, еще не готов немедленно его получить. Не исключено, что ему, к примеру, сначала нужно завершить выполнение текущей команды, для чего, может быть, потребуется использовать шину. Если в момент запроса прерывания запрещены, возможны еще большие задержки. В таком случае устройство должно дождаться готовности процессора и лишь после этого поместить данные на шину. Когда процессор готов получить код вектора прерывания, он активизирует линию подтверждения прерывания INTA. Устройство ввода-вывода отвечает на это отправкой кода вектора прерывания и выдачей сигнала INTR.

Вложенные прерывания

В разделе 4.2.1 было сделано предположение, что на время выполнения программы обработки прерывания все прерывания должны быть запрещены. При этом условии запрос одного устройства не сможет вызвать более одного прерывания. Этот принцип часто используется и в тех случаях, когда в системе имеется несколько устройств. В результате их прерывания обрабатываются по очереди, а начатая программа обработки прерывания выполняется до конца, до того как процессор получает запрос прерывания от другого устройства. Программы обработки прерываний в большинстве своем достаточно коротки, и вызываемая ими задержка для преобладающей части простых устройств обычно бывает вполне приемлемой.

Однако в некоторых случаях большая задержка с ответом на запрос прерывания может привести к неверному функционированию устройств. В качестве примера рассмотрим работу компьютера, отслеживающего время с помощью таймера

реального времени — устройства, которое направляет процессору запросы прерываний через фиксированные промежутки времени. По каждому из таких запросов процессор выполняет короткую программу обработки прерывания, увеличивающую хранящийся в памяти набор значений счетчиков, содержащих количество секунд, минут и т. д. Правильное функционирование таймера возможно при условии, что время задержки перед обработкой запроса прерывания значительно меньше временного интервала между запросами. Это требование будет выполняться лишь в том случае, если запрос прерывания от таймера будет приниматься во время выполнения программы обработки прерывания, вызванного другим устройством.

Последний пример показывает, что для правильной организации ввода-вывода должна использоваться система приоритетов устройств. Во время обслуживания процессором прерывания от устройства им должны приниматься запросы прерываний от устройств с более высоким приоритетом.

Многоуровневая система приоритетов означает, что в ходе выполнения программы обработки прерываний запросы на прерывания от одних устройств будут приниматься, а от других — нет. Для того чтобы реализовать такую схему обработки прерываний, необходимо процессору присвоить уровень приоритета, который будет меняться в зависимости от выполняемой программы. Уровень приоритета процессора — это уровень приоритета текущей выполняемой программы. Процессор принимает прерывания от устройств, имеющих более высокий приоритет, чем его собственный. Когда начинается выполнение программы обработки прерываний некоторого устройства, процессору назначается приоритет этого устройства. Тем самым запрещаются прерывания от любых устройств с тем же или более низким приоритетом.

Приоритет процессора обычно задается несколькими разрядами в слове, определяющем его состояние. Он может быть изменен при помощи программных команд, записывающих данные в регистр PS. Эти *привилегированные* команды выполняются лишь при условии, что процессор работает в режиме супервизора. (В данном режиме могут выполняться только программы операционной системы.) Перед началом реализации прикладных программ процессор переключается в пользовательский режим. Таким образом, пользовательская программа не может случайно или намеренно изменить приоритет процессора и нарушить работу системы. Попытка выполнить привилегированную команду в пользовательском режиме вызывает прерывание особого типа, которое называется *исключением защиты* (privilege exception) и описано в разделе 4.2.5.

Многоуровневая схема приоритетов может быть реализована с помощью отдельных линий запроса и подтверждения прерываний для каждого устройства, как показано на рис. 4.7. Каждой линии запроса прерывания присваивается свой уровень приоритета. Запросы прерываний, получаемые по этим линиям, направляются на арбитражную схему процессора. Запрос принимается только в том случае, если у него более высокий уровень приоритета, чем у процессора в данный момент.

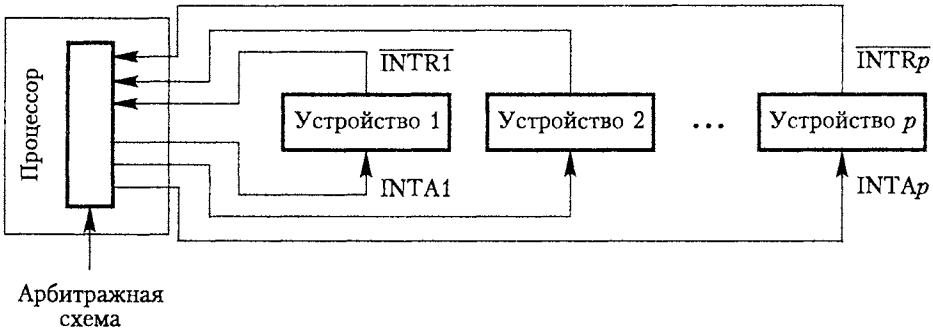


Рис. 4.7. Реализация приоритетов прерываний с использованием индивидуальных линий подтверждения прерывания

Одновременные запросы

Теперь давайте рассмотрим проблему одновременного поступления запросов прерываний от двух или более устройств. Процессор должен располагать средствами для выбора между этими запросами. При использовании схемы приоритетов, показанной на рис. 4.7, решение получается довольно простым: процессор выбирает запрос с наивысшим приоритетом. Но если несколько устройств используют одну линию запроса прерывания, как на рис. 4.6, то необходим иной механизм.

В таком случае проще всего опрашивать регистры состояния устройств ввода-вывода. Причем приоритеты этих устройств будут определяться порядком их опроса. При использовании векторных прерываний выбирается только одно устройство, которое должно отправить свой код вектора прерывания. Широко распространена схема соединения устройств в виде *гирляндной цепи* (рис. 4.8, *а*). При такой схеме линия запроса прерывания $\overline{\text{INTR}}$ является общей для всех устройств, а линия подтверждения прерывания INTA соединяет устройства в гирляндную цепь, так что сигнал по очереди проходит через каждое из них. Когда несколько устройств одновременно генерируют запрос прерывания и активизируется линия $\overline{\text{INTR}}$, процессор отвечает установкой сигнала на линии INTA в значение 1. Первым этот сигнал получает устройство 1. Если обслуживание ему не требуется, оно пересылает сигнал устройству 2. Если же устройство отправило запрос прерывания и ждет ответа, оно блокирует сигнал INTA и помещает свой идентификационный код на линии данных. Таким образом, в гирляндной схеме наивысший приоритет имеет устройство, которое ближе всего с точки зрения схемы подключения расположено к процессору.

Для реализации схемы, приведенной на рис. 4.8, *а*, требуется значительно меньше проводов, чем для отдельных соединений, показанных на рис. 4.7. Главным преимуществом схемы на рис. 4.7 является то обстоятельство, что процессор может выбирать устройства с учетом их приоритетов. Эти схемы можно объединить и в более универсальную структуру. На рис. 4.8, *б* устройства объединены в группы, каждой из которых назначен свой приоритет. Внутри группы устройства соединены в гирляндную цепь. Такая структура используется во многих компьютерных системах.

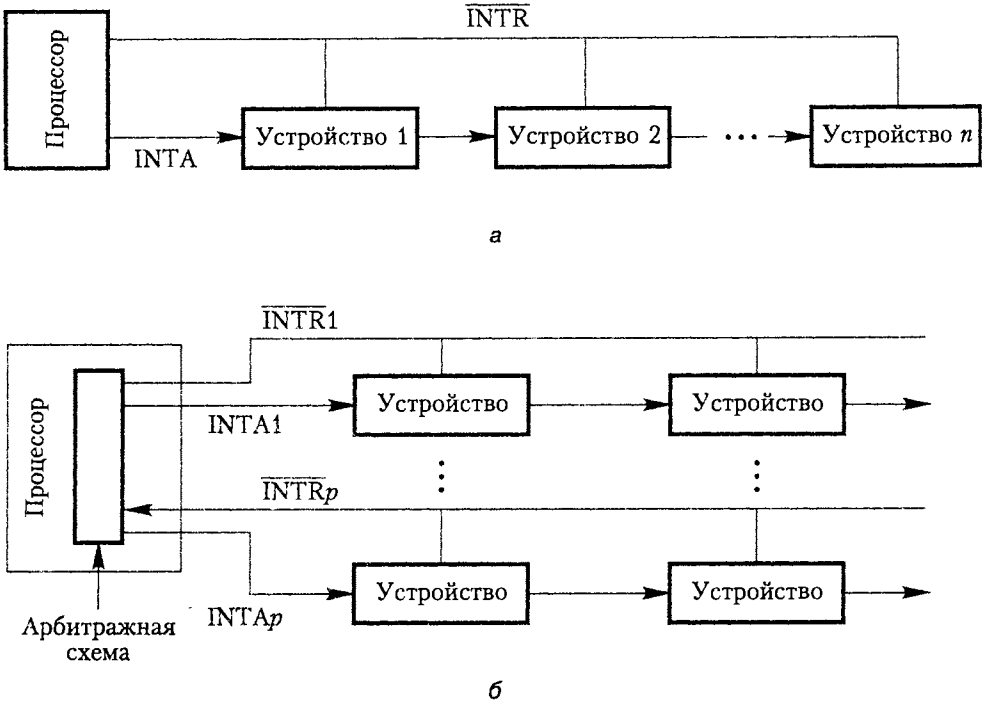


Рис. 4.8. Схемы приоритетов прерываний: гирляндная цепь (*а*); группы с приоритетами (*б*);

4.2.4. Управление запросами устройств

До сих пор мы предполагали, что интерфейс устройства ввода-вывода генерирует запрос прерывания, когда устройство готово к операции пересылки данных, например, когда флаг SIN на рис. 4.3 устанавливается в значение 1. При этом важно гарантировать, что запросы прерываний будут генерироваться только теми устройствами ввода-вывода, которые используются данной программой. Не задействованные на данный момент устройства не должны генерировать запросов прерываний, даже если они готовы выполнить операции ввода-вывода. Таким образом, в интерфейсных схемах устройств ввода-вывода должен быть заложен механизм, определяющий, когда можно, а когда нельзя генерировать запросы прерываний.

Обычно для этого используется один разряд, разрешающий или запрещающий прерывания, как, например, флаг разрешения прерывания от клавиатуры KEN и флаг разрешения прерывания от дисплея DEN в регистре CONTROL на рис. 4.3. Когда такой флаг установлен, устройство генерирует запрос прерывания, как только устанавливается соответствующий флаг в регистре STATUS. Одновременно интерфейсная схема устанавливает разряд KIRQ или DIRQ, указывая таким образом, что данное устройство запросило прерывание. Если флаг разрешения прерывания равен 0, интерфейсная схема не генерирует запросов прерываний независимо от того, какой флаг состояния активен.

Существует два независимых механизма управления запросами прерывания. Со стороны устройства решение о том, можно ли ему генерировать запросы прерывания, зависит от состояния разряда разрешения на прерывание. Со стороны процессора решение о том, будет ли принят запрос на прерывание, зависит либо от разряда разрешения прерываний в регистре PS, либо от системы приоритетов.

Пример 4.3

Рассмотрим процессор, в котором используется векторная схема прерываний. Начальный адрес подпрограммы обработки прерываний хранится в памяти по адресу INTVEC. Прерывания разрешаются установкой разряда разрешения прерываний IE (9 разряда) в слове состояния процессора. Клавиатура и дисплей, подключенные к этому процессору, имеют регистры состояния и данных, а также управляющий регистр (рис. 4.3).

Предположим, что в некоторой точке главной программы требуется прочитать введенную с клавиатуры строку и сохранить составляющие ее символы в последовательности байтов в памяти компьютера начиная с адреса LINE. Для выполнения данной операции с помощью прерываний нам нужно будет инициализировать процесс прерываний. Вот как это делается.

1. Загружаем начальный адрес программы обработки прерывания по адресу INTVEC.
2. Загружаем адрес LINE в память по адресу PNTR. Программа обработки прерываний будет использовать его в качестве указателя для сохранения вводимых символов в памяти.
3. Разрешаем прерывания от клавиатуры, для чего устанавливаем в 1 второй разряд регистра CONTROL.
4. Разрешаем прерывания в процессоре, для чего устанавливаем в 1 разряд IE в регистре состояния процессора PS.

После такой инициализации в ответ на ввод символа с клавиатуры интерфейсная схема клавиатуры будет генерировать запрос прерывания. Выполняемая в данный момент программа будет прервана и управление будет передано программе обработки прерываний, которая должна выполнить целый ряд действий.

1. Прочитать из регистра входных данных клавиатуры введенный символ. В ответ интерфейсная схема удалит свой запрос прерывания.
2. Сохранить символ в памяти компьютера по адресу, указанному в PNTR, и увеличить значение PNTR.
3. По достижении конца строки запретить прерывания от клавиатуры и проинформировать об этом программу Main.
4. Выполнить команду возврата из процедуры обработки прерывания.

Команды, ответственные за выполнение этой задачи, перечислены на рис. 4.9. Обнаружив конец строки, программа обработки прерывания очистит разряд KEN в регистре CONTROL, поскольку дальнейшего ввода не ожидается. А еще она установит в 0 переменную EOL (конец строки), которая первоначально имела значение 1. Мы предполагаем, что главная программа периодически проверяет данную переменную, чтобы узнать, готова ли введенная строка для обработки.

Главная программа

```
        Move        #LINE,PNTR      Инициализация указателя на буфер
        Clear       EOL             Очистка индикатора конца строки
        BitSet      #2,CONTROL      Разрешение прерываний от клавиатуры
        BitSet      #9,PS           Установка разряда разрешения прерываний
                                    в PS
        ⋮
```

Программа обработки прерывания

```
READ    MoveMultiple  R0-R1,-(SP)   Сохранение в стеке регистров R0 и R1
        Move          PNTR,R0       Загрузка указателя адреса
        MoveByte      DATAIN,R1     Чтение введенного символа
        MoveByte      R1,(R0)+      и сохранение его в памяти
        Move          R0,PNTR       Обновление указателя
        CompareByte   #$0D,R1       Проверка того, введен ли символ возврата
                                    каретки
        Branch≠0      RTRN
        Move          #1,EOL        Включение индикатора конца строки
        BitClear      #2,CONTROL    Запрет на прерывания от клавиатуры
RTRN    MoveMultiple  (SP)+,R0-R1   Восстановление регистров R0 и R1
        Return from interrupt       Возврат из прерывания
```

Рис. 4.9. Использование процедуры обработки прерываний для чтения строки символов с клавиатуры через регистры, показанные на рис. 4.3

Операции ввода-вывода в компьютерной системе обычно гораздо сложнее, чем в наших простых примерах. Как рассказывается в разделе 4.2.5, для пользовательских программ эти операции выполняет операционная система компьютера.

4.2.5. Исключения

Прерывание — это событие, которое приостанавливает выполнение текущей программы и запуск некоторой другой. До сих пор речь шла лишь о прерываниях, которые вызывались запросами, получаемыми в процессе ввода-вывода. Однако механизм прерываний используется и во множестве других ситуаций.

Любое событие, которое приводит к прерыванию, обычно называют *исключением*. Следовательно, примерами исключений могут служить и прерывания ввода-вывода. Ниже будет рассмотрен ряд других типов исключений.

Восстановление после ошибок

Правильная работа аппаратных компонентов компьютера обеспечивается множеством различных технологий. Например, в основной памяти ряда компьютеров содержится код контроля ошибок, позволяющий выявлять таковые в сохраняемых данных. Если произойдет ошибка, управляющие схемы обнаружат ее и проинформируют об этом с помощью прерывания процессор.

Процессор может прервать выполнение программы и в том случае, если в ходе реализации ее команд обнаружит ошибку или какую-либо нестандартную ситуацию. Например, заданный в команде код операции может не соответствовать ни одной из существующих команд, а арифметическая команда может попытаться осуществить деление на 0.

Если обработка исключения инициируется в результате подобной ошибки, процессор действует точно так же, как в случае запроса прерывания. Он приостанавливает выполнение текущей программы и запускает программу обработки исключения. Эта программа выполняет действия, необходимые для восстановления после ошибки (если это возможно), или информирует о ней пользователя. Как вы помните, в случае прерывания ввода-вывода процессор завершает выполнение текущей команды программы и лишь после этого начинает обработку прерывания. Однако если прерывание происходит из-за ошибки, выполнение текущей команды, как правило, завершить невозможно, поэтому процессор немедленно приступает к обработке исключения.

Отладка

Еще один важный тип исключений используется при отладке программ. Обычно в состав системного программного обеспечения входит программа под названием *отладчик,* помогающая программисту находить ошибки в коде. С помощью исключений отладчик реализует две важные функции: пошаговое выполнение программы (трассировку) и определение точек останова.

Когда процессор функционирует в режиме *трассировки,* после выполнения каждой команды программы происходит исключение, которое обрабатывается отладчиком. Отладчик дает возможность пользователю проанализировать содержимое регистров, памяти, различных устройств. По завершении работы отладчика выполняется следующая команда программы, после чего отладчик активизируется снова. Пока реализуется программа-отладчик, исключения трассировки запрещены.

Похожую возможность предоставляют программисту точки останова, с той лишь разницей, что выполняемая программа останавливается не после каждой команды, а только в определенных, выбранных им точках. Для этого обычно применяется команда, называемая ловушкой или программным прерыванием. Выполнение этой команды приводит к тем же результатам, что и получение запроса аппаратного прерывания. При отладке программы пользователь может прервать ее выполнение после команды i. Отладчик сохранит команду $i + 1$ и заменит ее командой программного прерывания. Когда выполняемая программа достигнет этой точки, она будет прервана и активизируется программа-отладчик. Это позволит пользователю проанализировать содержимое памяти и регистров. Когда пользователь будет готов продолжить выполнение отлаживаемой программы, отладчик восстановит сохраненную команду под номером $i + 1$ и выполнит команду возврата из прерывания.

Исключения защиты

Для защиты операционной системы компьютера от разрушения пользовательскими программами некоторые команды разрешено выполнять только тогда, когда

процессор работает в режиме супервизора. Такие команды называются *привилегированными*. Так, когда процессор работает в пользовательском режиме, он не выполняет команд, изменяющих уровень приоритета процессора или позволяющих пользовательской программе обращаться к тем областям памяти компьютера, которые выделены для других пользователей. Попытка выполнить такие команды приводит к исключению защиты, в ответ на которое процессор переходит в режим супервизора и начинает реализацию соответствующей подпрограммы операционной системы.

4.2.6. Прерывания в операционных системах

Операционная система (ОС) отвечает за координацию всех действий компьютера. Она выполняет операции ввода-вывода, взаимодействует с пользовательскими программами и управляет ими, интенсивно используя прерывания. Механизм прерываний позволяет операционной системе назначать приоритеты, переключаться от одной пользовательской программы к другой, реализовывать функции безопасности и защиты, координировать операции ввода-вывода. В данном разделе мы вкратце обсудим некоторые их этих аспектов.

В состав операционной системы входят программы обработки прерываний для всех подключенных к компьютеру устройств. Прикладные программы не выполняют операции ввода-вывода самостоятельно. Когда прикладной программе необходимо произвести одну из указанных операций, она указывает на подлежащие пересылке данные и просит операционную систему выполнить эту операцию. ОС приостанавливает выполнение программы и осуществляет запрошенную операцию ввода-вывода. По окончании операции ОС опять передает управление прикладной программе. (ОС и прикладная программа передают друг другу управление посредством программных прерываний.)

Операционная система выполняет для прикладных программ множество сервисных функций. Для их реализации многие процессоры поддерживают по нескольку разных команд программных прерываний, каждая их которых имеет собственный вектор прерывания. Эти команды могут использоваться для вызова разных частей ОС, в зависимости от выполняемой функции.

Если компьютер поддерживает два режима, супервизора и пользовательский, получив запрос прерывания, он всегда переключается в режим супервизора. Для этого он устанавливает соответствующий разряд регистра состояния процессора, предварительно сохранив в стеке старое его, регистра, значение. Таким образом, если прикладная программа вызывает ОС с помощью команды программного прерывания, процессор автоматически переключается в режим супервизора, предоставляя ОС полный доступ к ресурсам компьютера. Когда ОС выполняет команду возврата из прерывания, слово состояния процессора, соответствующее выполняемой прикладной программе, восстанавливается из стека. В результате процессор переключается обратно в пользовательский режим.

Чтобы проиллюстрировать взаимодействие между прикладными программами и операционной системой, давайте рассмотрим пример работы в *многозадачном режиме*, то есть когда процессор выполняет несколько пользовательских

программ одновременно. Применяемая при этом стандартная технология называется *квантованием времени*. Суть ее заключается в том, что каждая прикладная программа выполняется в течение короткого промежутка времени τ, называемого квантом времени, после чего другая программа выполняется в течение своего кванта времени и т. д. Величина τ определяется непрерывно работающими аппаратными часами (таймером), генерирующими прерывание каждые τ секунд.

На рис. 4.10 приведены программы, необходимые для реализации важнейших функций многозадачного окружения. Во время запуска операционной системы выполняется инициализационная подпрограмма, обозначенная на рисунке как OSINIT. Кроме всего прочего эта подпрограмма загружает векторы прерываний, расположенные по отведенным для них фиксированным адресам. В них она записывает начальные адреса программ обработки прерываний. В частности, OSINIT загружает начальный адрес программы-планировщика SHEDULER по адресу, соответствующему вектору прерывания таймера. Таким образом, в конце каждого кванта времени прерывание таймера приводит к выполнению этой программы.

Программа вместе с информацией, описывающей ее текущее состояние, определяется ОС как *процесс*. Процесс может находиться в одном из трех состояний: выполнение, ожидание и блокировка, или останов. Под выполняющимся процессом подразумевается выполняемая в данный момент программа. Готовый к выполнению процесс — это программа, ожидающая выбора планировщиком, реализация которой в любой момент может быть начата или продолжена. А заблокированный процесс — это программа, выполнение которой по какой-то причине в данный момент не может быть продолжено (скажем, из-за того, что она ожидает завершения запрошенной операции ввода-вывода).

Предположим, что в течение заданного кванта времени программа А находится в состоянии выполнения. В конце этого кванта времени таймер прерывает работу этой программы и запускает планировщик SHEDULER. Основная задача программы состоит в определении того, какая из пользовательских программ должна выполняться в течение следующего кванта времени. Она начинает свою работу с сохранения всей информации, которая потребуется позднее, при возобновлении работы программы А. Эта информация, называемая состоянием программы, включает содержимое регистров, счетчик команд и слово состояния процессора. Регистры должны быть сохранены потому, что они могут содержать промежуточные результаты вычислений, выполняющихся в момент окончания кванта времени. Счетчик команд указывает, с какой команды должно быть возобновлено выполнение программы. Слово состояния процессора содержит флаги условий и другую важную информацию, в том числе коды приоритетов.

Далее SHEDULER выбирает для выполнения какую-нибудь другую программу, предположим В, приостановленную ранее и находящуюся в режиме готовности. Планировщик восстанавливает всю информацию, сохраненную после приостановления программы В, включая содержимое регистров PS и PC, и выполняет команду возврата из прерывания. В результате этого выполнение программы В возобновляется и продолжается в течение τ секунд, после чего таймер снова генерирует прерывание и выполняется *переключение контекста* на другой процесс, находящийся в состоянии готовности.

OSINIT	Установка векторов прерываний:
	таймер квантования (SCHEDULER)
	программное прерывание (OSSERVICES)
	прерывания от клавиатуры (IOData)
	⋮
OSSERVICES	Анализ стека для определения запрошенной операции
	Вызов соответствующей программы
SCHEDULER	Сохранение состояния программы
	Выбор готового к выполнению процесса
	Восстановление сохраненного контекста нового процесса
	Проталкивание в стек новых значений PS и PC
	Возврат из прерывания

а

IOINIT	Перевод процесса в состояние блокировки
	Инициализация указателя на буфер в памяти и счетчика
	Вызов драйвера устройства для его инициализации и разрешения прерываний
	Возврат из подпрограммы
IODATA	Опрос устройств для определения инициатора прерывания
	Вызов соответствующего драйвера
	Перевод процесса в состояние готовности, если END=1
	Возврат из прерывания

б

KBDINIT	Разрешение прерываний
	Возврат из подпрограммы
KBDDATA	Проверка состояния устройства
	Если устройство готово к работе, пересылка символа
	Если значение символа равно CR, то флаг END устанавливается в 1, что соответствует запрету на прерывания; в противном случае END устанавливается в 0
	Возврат из подпрограммы

в

Рис. 4.10. Некоторые из программ операционной системы: программы инициализации ОС, сервисов и планировщика (*а*); программы ввода-вывода (*б*); драйвер клавиатуры (*в*)

Предположим, что программе А требуется прочитать вводимую с клавиатуры строку. Вместо того чтобы выполнять эту операцию самостоятельно, программа запрашивает у операционной системы функцию ввода-вывода. Для передачи

операционной системе информации о требуемой операции (в том числе об устройстве ввода-вывода и адресе буфера в области данных программы, в который следует поместить прочитанную строку) она использует стек или регистры процессора. Затем программа выполняет команду программного прерывания. Вектор прерывания этой команды указывает на подпрограмму OSSERVICES, представленную на рис. 4.10, *а*. Эта подпрограмма анализирует информацию в стеке и инициирует запрошенную операцию с помощью вызова соответствующей подпрограммы ОС. В нашем примере она вызывает подпрограмму IOINIT, отвечающую за инициирование операций ввода-вывода (рис. 4.10, *б*).

Пока выполняется операция ввода-вывода, запросившая ее программа не может продолжать свою работу. Поэтому подпрограмма IOINIT переводит процесс, связанный с программой А, в состояние блокировки, указывающее планировщику, что пока выполнение этой программы не может быть продолжено. Затем данная подпрорамма производит подготовительную работу, необходимую для выполнения операции ввода-вывода (например, инициализацию указателей и счетчика байтов), и вызывает подпрограмму, осуществляющую операцию ввода-вывода.

Как правило, операционные системы строятся так, чтобы все программное обеспечение, связанное с конкретным устройством, заключалось в отдельный модуль, называемый *драйвером устройства*. Такой модуль легко добавить в операционную систему и легко из нее удалить. Мы предположили, что драйвер клавиатуры состоит из двух подпрограмм, KBDINIT и KBDDATA (рис. 4.10, *в*). Подпрограмма IOINIT вызывает KBDINIT, выполняющую инициализацию устройства и его интерфейсной схемы. Кроме того, подпрограмма KBDINIT разрешает прерывания от данного устройства, установив соответствующий бит в управляющем регистре его интерфейсной схемы, а затем возвращает управление подпрограмме IOINIT, которая, в свою очередь, возвращает управление подпрограмме OSSERVICES. После этого клавиатура готова к операциям пересылки данных. После каждого нажатия клавиши она будет генерировать запрос прерывания.

После возврата управления подпрограмме OSSERVICES планировщик SHEDULER выбирает для выполнения другую пользовательскую программу. Он не сможет выбрать программу А, поскольку она пока находится в заблокированном состоянии. Команда возврата из прерывания, после которой начинается выполнение выбранной пользовательской программы, разрешает прерывания в процессоре, загрузив в его регистр состояния новое содержимое. Таким образом, сгенерированный клавиатурой запрос прерывания будет принят процессором. Вектор этого прерывания указывает на программу ОС, названную IODATA. Поскольку к одной линии запроса прерывания может быть подсоединено несколько устройств, программа IODATA начинает опрашивать эти устройства, чтобы узнать, кому из них потребовалось обслуживание. Затем она вызывает соответствующий драйвер устройства для обслуживания запроса. В нашем случае будет вызван драйвер KBDDATA, который перешлет в процессор один символ. Если это символ возврата каретки, драйвер установит значение флага END в 1, чтобы проинформировать IODATA о завершении операции ввода-вывода. После этого программа IODATA изменит состояние процесса А — теперь он будет не блокированным, а готовым к выполнению, и когда подойдет его очередь, планировщик сможет выбрать его для выполнения в течение очередного кванта времени.

4.3. Примеры обработки прерываний различными процессорами

В предыдущем разделе было рассказано об основных принципах организации и обработки прерываний. Имеющиеся на рынке процессоры обычно поддерживают большую часть рассмотренных нами функций и механизмов, хотя отдельные методы управления прерываниями ими могут и не поддерживаться. В частности, процессор может поддерживать векторные прерывания, ускоряющие вызов программ обработки прерываний от конкретных устройств. А задача идентификации устройства и определения начального адреса соответствующей программы обработки прерываний в качестве альтернативы может быть возложена на программное обеспечение, которое должно будет прибегнуть к опросу всех устройств. В следующих разделах рассказывается о механизмах обработки прерываний, используемых в трех процессорах, которые описаны в главе 3.

4.3.1. Механизм прерываний процессора ARM

Механизм прерываний процессора ARM отличается простотой и мощью. В нем определено пять источников прерываний, из которых только двум соответствуют внешние линии запроса прерывания, IRQ и FIQ (Fast Interrupt Request — быстрый запрос прерывания). Имеется одна команда программного прерывания, SWI, и два типа прерываний, которые могут быть вызваны нестандартными условиями, возникающими в ходе выполнения программ. Это прерывания, вызываемые внешним сбоем, за которым следует ошибка шины, и прерывания, обусловленные попыткой выполнить несуществующую команду. Исключения обрабатываются в соответствии с иерархией приоритетов:

1. Перезапуск (Reset) — наивысший приоритет;
2. Ошибка данных (Data abort);
3. Быстрый запрос прерываний (FIQ);
4. Запрос прерываний (IRQ);
5. Сбой выборки команды (Prefetch abort);
6. Несуществующая команда (Undefined instruction) — низший приоритет.

Условие Reset включено в эту иерархию по той простой причине, что оно имеет естественный приоритет перед всеми остальными условиями и должно вызывать немедленный перевод процессора в определенное начальное состояние. Условие Data abort обусловлено возникновением ошибки при чтении или записи данных, а условие Prefetch abort — ошибки во время выборки команд из памяти.

На рис. 3.1 показана структура регистра состояния процессора ARM, называемого CPSR (Current Program Status Register — регистр текущего состояния программы). Младший байт данного регистра состояния показан на рис. 4.11. Он содержит два разряда маскирования прерываний, по одному для прерываний IRQ и FIQ. Когда любой из этих разрядов равен 1, соответствующее прерывание запрещено. Кроме того, в регистре имеется пять разрядов режима, M_{4-0}, которые определяют режим работы процессора. Процессор ARM может работать в шести

режимах: пользовательском и пяти привилегированных, по одному для каждого из пяти типов исключений.

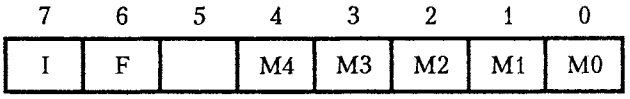


Рис. 4.11. Младший байт регистра состояния процессора ARM

Когда процессор переключается из одного режима в другой, он переключает и несколько доступных программе регистров. Какие именно регистры доступны в каждом из режимов, показано на рис. 4.12. Регистры от R0 до R7, R15 (PC) и CPSR доступны во всех режимах. В привилегированных режимах, за исключением режима FIQ, доступны также регистры от R8 до R12. А вот регистры R13 и R14 в режимах IRQ, Supervisor, Abort и Undefined заменяются новыми регистрами. В режиме FIQ вместо регистров R8—R14 используются регистры R8_fiq—R14_fiq. Регистры, заменяющие регистры пользовательского режима, называются *запасными* (banked registers). Они могут использоваться программами обработки прерываний без необходимости сохранения содержимого заменяемых ими регистров, применяемых в пользовательском режиме. Например, когда в режиме IRQ процессору встречается команда со ссылкой на регистр R13, он обращается к регистру R13_irq, а не к регистру R13, к которому ему пришлось бы обратиться в пользовательском режиме. Для каждого режима, кроме пользовательского, имеется специальный регистр (SPSR_svc, SPSR_irq и т. д.), предназначенный для сохранения содержимого регистра CPSR на время обработки прерывания. Он называется регистром сохранения состояния процессора (saved processor status register).

Программы обработки прерывания начинаются в памяти по фиксированным адресам (табл. 4.1). После прерывания процессор переходит в нужный режим и начинает выполнять программу, расположенную по соответствующему адресу. Поскольку по этому адресу выделяется место только для одной команды программы (если не считать команду прерывания FIQ), там хранится команда перехода к программе обработки прерывания. Программа обработки прерывания FIQ может располагаться и по адресу, заданному в векторе прерывания.

Таблица 4.1. Адреса векторов прерывания для процессора ARM

Адрес (шестнадцатеричный)	Исключение	Режим процессора
0	Сбос	Супервизор
4	Несуществующая команда	Не определен
8	Программное прерывание	Супервизор
C	Сбой при выборке команды	Аварийное завершение
10	Ошибка данных	Аварийное завершение
14	Зарезервировано	
18	IRQ	IRQ
1C	FIQ	FIQ

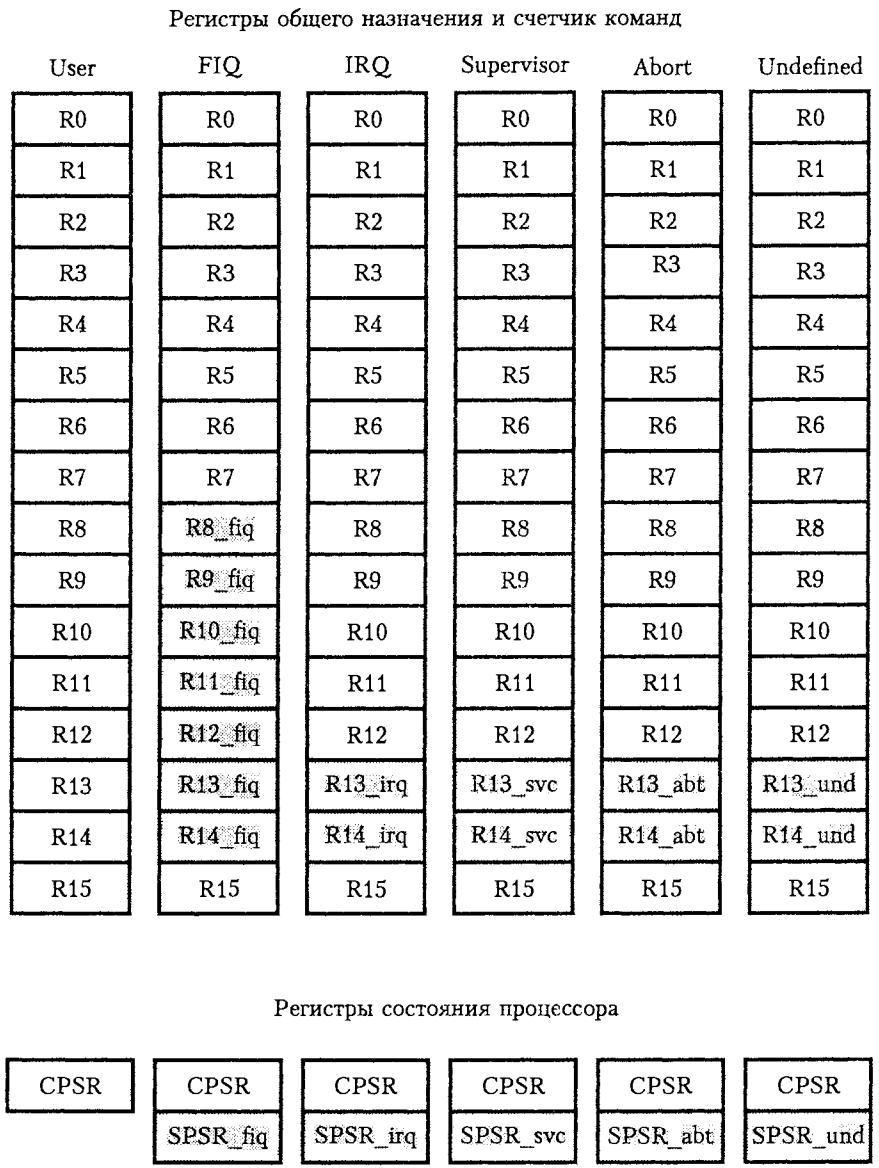


Рис. 4.12. Регистры, доступные в разных режимах процессора ARM

Когда процессор принимает запрос на прерывание, он выполняет ряд следующих действий.

1. Сохраняет адрес возврата прерванной программы в регистре 14 соответствующего режима. Например, в случае прерывания FIQ адрес возврата сохраняется в регистре R14_fiq. Как будет рассказано ниже, сохраняемое значение зависит от типа исключения.

2. Сохраняет содержимое регистра состояния процессора CPSR в соответствующем регистре SPSR.
3. Изменяет разряды режима в регистре CPSR в соответствии с типом прерывания (см. последний столбец табл. 4.1). В случае прерываний IRQ и FIQ разряд маскирования устанавливается в 1, в результате чего дальнейшие прерывания по этой же линии запрещаются.
4. Переходит к программе обработки прерывания, которая начинается по адресу, указанному в соответствующем векторе прерывания.

В процессоре ARM используется конвейерный принцип обработки команд. Это означает, что очередная команда считывается из памяти в процессор еще до окончания выполнения предыдущей команды (более подробно об этом будет рассказано в главе 8). Предположим, процессор выбирает команду I_1, хранящуюся по адресу A. Он увеличивает содержимое регистра PC до A+4 и начинает выполнение команды I_1. Еще до завершения этой операции он выбирает из памяти команду I_2, расположенную по адресу A+4, и тут же увеличивает содержимое PC до A+8. Теперь предположим, что, когда команда I_1 уже почти выполнена, процессор обнаруживает запрос прерывания IRQ и начинает производить описанные выше действия. Он копирует содержимое регистра PC, теперь уже равное A+8, в регистр R14_irq. Команда I_2, которая уже извлечена из памяти, но еще не выполнена, пока уничтожается в процессоре (но не в памяти). Именно с нее будет продолжена работа после возврата управления из программы обработки прерывания.

Итак, при вызове программы обработки прерывания в регистре R14_irq сохраняется адрес A+8, тогда как адресом возврата из этой программы должен быть A+4. Это означает, что перед использованием содержимого регистра R14_irq в качестве адреса возврата программа обработки прерывания должна вычесть из него 4. То есть команда возврата из прерывания должна загрузить в регистр PC значение [R14_irq] – 4. Кроме того, она должна скопировать содержимое регистра SPSR_irq в CPSR. Последнее действие возвращает процессор в тот режим, в котором он работал до прерывания, и очищает маску прерывания, чтобы тут же снова разрешить таковое. Эти действия выполняются командой

```
SUBS  PC,R14_irq,#4
```

которая вычитает из содержимого регистра R14_irq значение 4 и сохраняет результат в регистре R15. Суффикс S в имени команды означает, что процессор должен установить коды условий. Когда целевым регистром команды является PC, суффикс S также означает, что процессор должен скопировать содержимое регистра SPSR_irq в CPSR, завершая действия, необходимые для возврата в прерванную программу.

Значение, которое нужно вычесть из содержимого регистра R14 для получения правильного адреса возврата, зависит от особенностей выполнения команды в конвейере процессора. Для разных типов исключений оно может быть разным. Например, в случае программного прерывания, вызванного командой SWI, сохраненное в регистре R14_irq значение является правильным адресом возврата.

Поэтому возврат из программы обработки прерывания SWI может быть выполнен с помощью команды

```
MOVS  PC,R14_svc
```

Для каждого из исключений, перечисленных в табл. 4.1, в табл. 4.2 приведены правильные значения адресов возврата и команды, с помощью которых выполняется возврат в прерванную программу. Обратите внимание, что для прерывания Abort, которое может быть обусловлено сбоем в работе шины, правильным адресом возврата является адрес вызвавшей ошибку команды. Предполагается, что управляющее программное обеспечение может попытаться еще раз выполнить эту команду.

При работе в привилегированном режиме две специализированные команды MOV, называемые MSR и MRS, пересылают данные между заданным регистром и текущим или сохраненным регистром PSR. Например, команда

```
MRS  R0,CPSR
```

копирует содержимое регистра CPSR в R0. Аналогичным образом команда

```
MSR  SPSR,R0
```

загружает регистр SPSR из регистра R0. Эти команды выполняются в тех случаях, когда операционная система должна разрешать и запрещать прерывания, как в примере 4.4, приведенном далее в этом разделе.

Таблица 4.2. Корректные адреса возврата из исключений

Исключение	Сохраненный адрес	Правильный адрес возврата	Команда возврата
Несуществующая команда	PC+4	PC+4	MOVS PC,R14_und
Программное прерывание	PC+4	PC+4	MOVS PC,R14_svc
Сбой при выборке команды	PC+4	PC	SUBS PC,R14_abt,#4
Ошибка данных	PC+4	PC	SUBS PC,R14_abt,#8
IRQ	PC+4	PC	SUBS PC,R14_irq,#4
FIQ	PC+4	PC	SUBS PC,R14_fiq,#4

В регистре PC находится адрес команды, вызвавшей исключение. Для IRQ и FIQ это адрес первой команды, не выполненной из-за прерывания.

Стеки и вложенные вызовы

Механизм прерываний процессора ARM позволяет сохранить адрес возврата в регистре и не имеет встроенной поддержки стека для вложенных вызовов подпрограмм и прерываний. Зато он предоставляет программисту возможность самостоятельно реализовать эти функции, причем только в том случае, если они действительно нужны. Если прерывания генерируются разными источниками, не исключено, что они являются вложенными. Например, программа обработки

прерывания IRQ, сохранившая адрес возврата в регистре R14_irq, может быть прервана прерыванием FIQ, имеющим более высокий приоритет. Новый адрес возврата будет сохранен в регистре R14_fiq.

Для того чтобы компьютер мог обрабатывать вложенные прерывания от одного и того же источника, содержимое регистров R14 и SPSR должно сохраняться в стеке. Эту операцию можно выполнить программным путем, используя в качестве указателя стека регистр R13. Предназначенные для этой цели специализированные регистры R14 и R13 доступны в любом режиме. Программа обработки прерываний может сохранить содержимое регистров R14 и SPSR в собственном стеке, а затем очистить маску прерывания в CPSR. Кроме того, она может сохранить в стеке содержимое других регистров, чтобы создать дополнительное рабочее пространство для себя. Для быстрого прерывания FIQ доступны предназначенные специально для этой цели регистры R8_fiq и R13_fiq, которыми процессор может пользоваться сразу, не сохраняя их содержимое в стеке.

В главе 3 рассказывалось, что команды LDM и STM, выполняющие пересылку нескольких слов, удобны для выполнения операций со стеком. Например, используя в качестве указателя стека регистр R13, подпрограмма или программа обработки прерывания может сохранить несколько регистров и адрес возврата с помощью такой команды:

```
STMFD  R13!,{R0,R1,R2,R14}
```

Аналогичным образом команду LDMFD можно применить для восстановления сохраненных значений. В случае прерывания SWI или исключения при выборке команды сохраненное значение регистра R14 является корректным адресом возврата. Поэтому его можно непосредственно восстановить в регистр R15 и тем самым вызвать возврат в прерванную программу:

```
LDMFD  R13!,{R0,R1,R2,R15}^
```

Символ «^» в конце команды вызывает тот же эффект, что и суффикс S в использовавшейся выше команде SUBS. Он заставляет процессор одновременно с загрузкой регистра R15 скопировать содержимое регистра SPSR в CSPR. Обратите внимание, что для возврата из прерываний IRQ и FIQ команда LDM использоваться не может, поскольку перед ее выполнением содержимое регистра R14 должно быть изменено в соответствии с табл. 4.2.

Пример 4.4

На рис. 4.13 дан пример использования прерываний, посредством которого мы хотим продемонстрировать, как переписать для процессора ARM программу, приведенную на рис. 4.9. При этом мы делаем три следующих предположения: клавиатура соединена с линией прерывания IRQ, а соответствующий вектор прерывания содержит команду перехода к подпрограмме READ; при загрузке данного фрагмента кода в память по адресу PNTR загружается адрес буфера LINE; адреса PNTR и EOL очень близки, поэтому доступ к ним может осуществляться с помощью относительного метода адресации. Программа Main позволяет осуществлять прерывания в интерфейсе клавиатуры и процессоре, для чего она устанавливает флаг KEN в регистре клавиатуры CONTROL и очищает маску I в регистре

состояния процессора. Маска I, хранящаяся в разряде 7, очищается путем загрузки в регистр CPSR с помощью команды MSR значения $50.

В программе обработки прерывания мы используем команды LDM и STM для сохранения и восстановления регистров, а команду SUBS — для возврата в прерванную программу. Адрес регистра клавиатуры DATAIN (рис. 4.3) загружается в регистр процессора с помощью команды ADR, описанной в разделе 3.4.1. Мы предполагаем, что адресом управляющего регистра CONTROL является DATAIN+3.

Главная программа

```
        MOV     R0,#0
        STR     R0,EOL          Очистка флага EOL
        ADR     R1,DATAIN       Загрузка адреса в регистр DATAIN
        LDRB    R0,[R1,#3]      Чтение содержимого регистра CONTROL
        ORR     R0,R0,#4        Установка разряда KEN в регистре
        STRB    R0,[R1,#3]      CONTROL с целью разрешить
                                прерывания от клавиатуры
        MOV     R0,#&50         Разрешение прерываний IRQ
        MSR     CPSR,R0         в процессоре и переключение
                                в пользовательский режим
        ⋮
```

Программа обработки прерываний IRQ

```
READ    STMFD   R13!,{R0-R2,R14_irq} Сохранение содержимого регистров R0,
                                     R1 и R14_irq в стеке
        ADR     R1,DATAIN       Загрузка адреса регистра DATAIN
        LDRB    R0,[R1]         Считывание введенного символа
        LDR     R2,PNTR         Загрузка значения указателя
        STRB    R0,[R2],#1      Сохранение символа и увеличение
                                значения указателя
        STR     R2,PNTR         Обновление значения указателя в памяти
        CMPB    R0,#&0D         Проверка того, введен ли символ
                                возврата каретки
        LDMNEFD R13!,{R0-R2,R14_irq} Если нет, восстановление регистров
        SUBNES  PC,R14_irq,#4   и выход из программы
        LDRB    R0,[R1,#3]      В противном случае считывание
                                содержимого регистра CONTROL
        AND     R0,R0,#&FB      Очистка разряда KEN с целью
        STRB    R0,[R1,#3]      запретить прерывания от клавиатуры
        MOV     R0,#1           Установка флага EOL
        STR     R0,EOL
        LDMFD   R13!,{R0-R2,R14} Восстановление регистров
        SUBS    PC,R14_irq,#4   и выход из программы
```

Рис. 4.13. Программа обработки прерывания для процессора ARM, считывающая вводимую с клавиатуры строку (основой служит программа, представленная на рис. 4.9)

4.3.2. Механизм прерываний процессора 68000

Процессор 68000 имеет восемь уровней приоритета прерываний. Текущий приоритет процессора кодируется тремя разрядами в его слове состояния (рис. 4.14). Самый низкий приоритет обозначен номером 0. Устройства ввода-вывода соединяются с процессором 68000 по такой же схеме, как на рис. 4.8, *б* — в данном случае прерываниям назначены приоритеты от 1 до 7. Запрос прерывания принимается только тогда, когда его приоритет выше, чем у процессора. Правда, имеется одно исключение: запрос с приоритетом 7 принимается всегда. Такой тип прерывания, управляемого фронтом сигнала, называется *немаскируемым*. Когда процессор принимает запрос прерывания, перед выполнением программы обработки прерывания уровень приоритета, указанный в регистре PS, автоматически повышается до уровня приоритета данного прерывания. Таким образом, запросы прерываний с тем же или более низким приоритетом запрещаются, за исключением прерываний с приоритетом 7, которые, как уже было сказано, разрешены всегда.

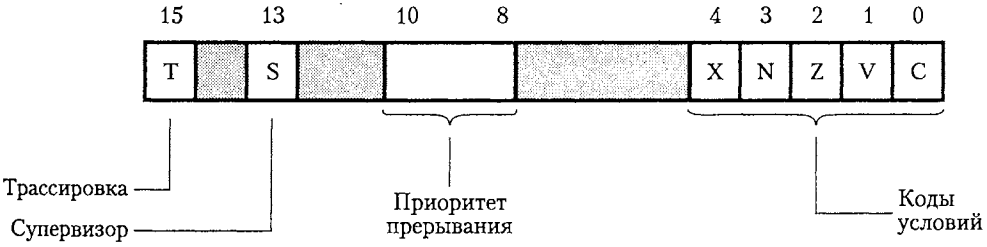


Рис. 4.14. Регистр состояния процессора 68000

Когда происходит прерывание, процессор автоматически сохраняет содержимое счетчика команд и слово состояния. Содержимое регистра PC проталкивается в стек процессора следом за содержимым регистра PS, а в качестве указателя стека используется регистр A7. Команда возврата из прерывания, в языке ассемблера процессора 68000 называемая RTE (Return From Exception), выталкивает верхний элемент стека в регистр PS, а следующий элемент — в регистр PC. Как видно на рис. 4.14, в регистре PS имеются разряды S (Supervisor) и T (Trace). Разряд S определяет, в каком режиме, то есть супервизора (S=1) или пользователя (S=0), работает процессор. Разряд T разрешает прерывания трассировки, о которых рассказывалось в разделе 4.2.4. Эта информация автоматически сохраняется процессором, когда он принимает запрос прерывания, и восстанавливается по окончании обработки такового. Если требуется сохранить еще какую-либо информацию, предположим, содержимое регистров общего назначения, это делается явно в программе обработки прерывания.

В процессоре 68000 используются векторные прерывания. Когда процессор принимает запрос прерывания, он считывает начальный адрес программы обработки такового из вектора прерывания, хранящегося в основной памяти компьютера. У него 256 векторов прерываний, пронумерованных цифрами от 0 до 255. Каждый вектор состоит из 32 разрядов, составляющих начальный адрес программы. Когда устройство запрашивает прерывание, оно может указать процессору,

какой вектор должен использоваться для его обработки. Для этого в ответ на сигнал подтверждения прерывания устройство направляет процессору 8-разрядный номер данного вектора. В качестве альтернативы процессор 68000 поддерживает функцию *автоматического выбора вектора прерывания*. Вместо того чтобы направить процессору номер вектора, устройство может активизировать специальную управляющую линию шины, указывающую, что оно желает использовать эту функцию. Тогда процессор сам выбирает один из семи предназначенных для этой цели векторов прерываний в соответствии с уровнем приоритета запрошенного прерывания.

Пример 4.5

Пример использования прерываний в процессоре 68000 показа на рис. 4.15. Эта программа аналогична приведенной на рис. 4.9, но написана специально для процессора 68000. Предполагается, что интерфейс клавиатуры использует функцию автоматического выбора вектора прерывания и генерирует запросы прерываний с приоритетом 2. Для того чтобы разрешить прерывания, процессор должен перейти в режим работы с более низким приоритетом (то есть меньше 2). Когда в регистр PS загружается битовая маска $100, приоритет процессора устанавливается в 1.

```
Главная программа
        MOVE.L    #LINE,PNTR      Инициализация указателя на буфер
        CLR       EOL             Очистка индикатора конца строки
        ORI.B     #4,CONTROL      Установка разряда KEN
        MOVE      #$100,SR        Установка для процессора приоритета
                                  равным 1
Программа обработки прерываний
READ    MOVEM.L   A0/D0,-(A7)     Сохранение регистров A0, D0 в стеке
        MOVEA.L   PNTR,A0         Загрузка указателя адреса
        MOVE.B    DATAIN,D0       Считывание введенного символа
        MOVE.B    D0,(A0)+        Сохранения символа в буфере памяти
        MOVE.L    A0,PNTR         Обновление указателя
        CMPI.B    #$0D,D0         Проверка того, введен ли символ возврата
                                  каретки
        BNE       RTRN
        MOVE      #1,EOL          Установка индикатор конца строки
        ANDI.B    #$FB,CONTROL    Очистка разряда KEN
RTRN    MOVEM.L   (A7)+,A0/D0     Восстановление регистров A0, D0
        RTE
```

Рис. 4.15. Программа обработки прерывания для процессора 68000, считывающая вводимую с клавиатуры строку (основой служит программа, представленная на рис. 4.9)

4.3.3. Механизм прерываний процессора Pentium

Примерами процессоров архитектуры IA-32, могут служить процессоры Pentium, в которых используются две линии запроса прерывания, а именно NMI (Non-Maskable Interrupt) — для немаскируемых прерываний и INTR — для маскируемых прерываний, также называемых пользовательскими. Запросы прерываний по линии NMI всегда принимаются процессором. Запросы по линии INTR принимаются только в том случае, если они имеют более высокий уровень приоритета, чем текущая выполняемая программа (об этом речь пойдет чуть позже). Прерывания INTR можно разрешать или запрещать, устанавливая разряд разрешения прерываний в регистре состояния процессора.

В дополнение к внешним прерываниям существует множество исключений, связанных с событиями, которые происходят во время выполнения программ. К числу таких событий относятся, в частности, неверные коды операций, ошибки деления, переполнения и многие другие.

Любое из указанных событий заставляет процессор передать управление программе обработки прерывания. Каждому прерыванию или исключению назначается номер вектора. В случае прерываний по линии INTR номер вектора направляется устройством ввода-вывода процессору по шине после подтверждения факта прерывания. Для остальных исключений номер вектора задается заранее. Зная номер вектора, процессор определяет начальный адрес программы обработки прерывания, для чего он обращается к таблице, называемой Interrupt Descriptor Table (таблица дескрипторов прерываний).

Процессор Pentium работает совместно с микросхемой, называемой APIC (Advanced Programmable Interrupt Controller — усовершенствованный программируемый контроллер прерываний). Через нее различные устройства ввода-вывода соединяются с процессором. Контроллер прерываний определяет приоритеты устройств и направляет процессору соответствующий каждому из них номер вектора прерывания.

В литературе, посвященной процессорам Intel, регистр состояния процессора (рис. 3.37) называется EFLAGS. На рис. 4.16 показаны семь разрядов этого регистра, с 8 по 15, где содержатся флаг разрешения прерываний IF (Interrupt enable Flag), флаг трассировки TF (Trap Flag), а также уровень привилегий ввода-вывода IOPL (I/O Privilege Level). Когда IF = 1, прерывания по линии INTR разрешены. Флаг трассировки разрешает прерывания трассировки после каждой команды программы.

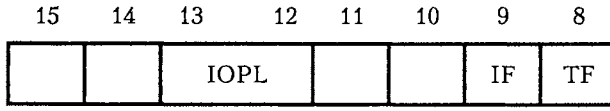


Рис. 4.16. Часть регистра состояния процессора Pentium

У процессора Pentium довольно сложная структура приоритетов, в соответствии с которой программы операционной системы условно делятся на четыре группы

и каждой из них назначается один из четырех уровней приоритета. Для каждого из таких уровней выделяется свой сегмент адресного пространства процессора. При переключении от одного уровня привилегий к другому выполняется множество проверок, реализующих так называемый механизм блокировки. Такая архитектура позволяет разрабатывать операционные системы с очень высокой степенью защиты. Но процессор Pentium может работать и в очень простом режиме, когда вообще не существует никаких привилегий и все программы функционируют в одном сегменте памяти. Такой простой режим мы и рассмотрим в данном разделе.

При возникновении исключения, а также после получения запроса прерывания процессор выполняет операции, перечисленные ниже.

1. Проталкивает в стек процессора, на который указывает регистр ESP, значение регистра текущего сегмента CS (Current Segment) и указателя команды EIP.
2. Если исключение вызвано нестандартной ситуацией при выполнении программы, помещает в стек код, определяющий причину исключения.
3. Если нужно, очищает флаг разрешения прерывания, чтобы дальнейшие прерывания от того же источника были запрещены.
4. По значению вектора прерывания находит в таблице дескрипторов прерываний начальный адрес программы обработки прерывания и загружает его в EIP, после чего продолжает выполнение команд.

Обслужив запрос (например, выполнив пересылку входных или выходных данных), программа обработки прерывания возвращает управление прерванной программе, для чего она выполняет команду возврата из прерывания IRET. Эта команда восстанавливает из стека значения регистров EIP и CS и регистра состояния процессора, восстанавливая тем самым состояние процессора.

Подобно любой подпрограмме программа обработки прерывания может создать для себя временное рабочее пространство, сохранив в стеке значения регистров. В этом случае должна быть гарантия того, что перед выполнением команды IRET программы указатель стека ESP будет содержать адрес возврата.

Пример 4.6

На рис. 4.17 вы видите ту же программу, которая был представлена на рис. 4.9, но переписанную для процессора Pentium. Здесь мы сделали предположение, что клавиатура направляет запрос прерывания с номером вектора 32 и что соответствующая запись в таблице дескрипторов прерываний содержит начальный адрес программы обработки прерывания READ. Для того чтобы разрешить прерывания в процессоре, нужно выполнить команду STI, устанавливающую флаг IF в регистре состояния процессора в 1.

```
Главная программа
        MOV     EOL,0
        MOV     BL,4
        OR      CONTROL,BL      Установка разряда KEN для разрешения
                                прерываний от клавиатуры
        STI                     Установка флага прерываний в регистре
                                процессора
Программа обработки прерываний
READ    PUSH    EAX             Сохранение регистра EAX в стеке
        PUSH    EBX             Сохранение регистра EBX в стеке
        MOV     EAX,PNTR        Загрузка указателя адреса
        MOV     BL,DATAIN       Считывание введенного символа
        MOV     [EAX],BL        Сохранение символа
        INC     DWORD PTR [EAX] Увеличение значения указателя
        CMP     BL,0DH          Проверка того, введен ли символ возврата
                                каретки
        JNE     RTRN
        MOV     BL,4
        XOR     CONTROL,BL      Очистка разряда KEN
        MOV     EOL,1           Установка флага EOL
RTRN    POP     EBX             Восстановление регистра EBX
        POP     EAX             Восстановление регистра EAX
        IRET
```

Рис. 4.17. Программа обработки прерывания для процессоров IA-32, считывающая вводимую с клавиатуры строку (основой служит программа, показанная на рис. 4.9)

4.4. Прямой доступ к памяти

В предыдущем разделе мы пытались сконцентрировать ваше внимание на пересылке данных между процессором и устройствами ввода-вывода. Для этой цели используются команды типа

```
Move  DATAIN,R0
```

Данная команда выполняется лишь после того, как процессор определит, что устройство ввода-вывода готово к очередной операции. Для этого процессор опрашивает флаг состояния в интерфейсе устройства или ждет, пока устройство само направит ему запрос на прерывание. В любом случае производится много лишней работы, поскольку для каждого пересылаемого слова данных выполняется несколько программных команд. Однако команды нужны не только для опроса регистра состояния, но и для увеличения адреса в памяти и отслеживания количества слов. При использовании прерываний издержки еще больше, поскольку

приходится сохранять и восстанавливать значение счетчика команд и другую информацию о состоянии.

Поэтому для быстрой пересылки больших блоков данных применяется другой подход. Компьютер может содержать специальное управляющее устройство, позволяющее пересылать блоки данных между внешним устройством и основной памятью без постоянного участия процессора. Эта технология называется *прямым доступом к памяти* (*ПДП*), по-английски — Direct Memory Access (DMA).

Операции ПДП выполняются управляющей схемой, входящей в состав интерфейса устройства ввода-вывода. Эта схема называется *контроллером ПДП*. Контроллер ПДП выполняет ту же задачу, что и процессор, обращающийся к основной памяти. Для каждого пересылаемого слова он генерирует адрес памяти и сигналы шины, управляющие пересылкой данных. Поскольку контроллер ПДП производит пересылку блоков данных, он сам увеличивает адрес, по которому будет записываться каждое следующее слово, и отслеживает количество таких операций.

Хотя контроллер ПДП работает без участия процессора, он управляется выполняемой процессором программой. В частности, чтобы инициировать пересылку блока слов, процессор пересылает контроллеру начальный адрес этого блока, сведения о количестве составляющих его слов и направлении пересылки. Получив такую информацию, контроллер приступает к выполнению операции. По окончании пересылки он информирует об этом процессор с помощью сигнала прерывания.

Пока контроллер ПДП производит пересылку данных, запросившая ее программа не может продолжать свою работу, и процессор часто используется для выполнения другой программы. По окончании пересылки процессор может вернуться к исходной программе.

Операции ввода-вывода всегда выполняются операционной системой компьютера в ответ на запрос прикладной программы. Операционная система отвечает за приостановку выполнения одной программы и активизацию другой. Поскольку операции ввода-вывода осуществляются с использованием прямого доступа к памяти, ОС переводит запросившую такую операцию программу в режим блокировки (см. раздел 4.2.6), инициирует операцию ПДП и начинает выполнение другой программы. По завершении пересылки контроллер ПДП направляет процессору запрос прерывания. В ответ ОС переводит приостановленную программу в режим готовности, чтобы ее мог выбрать планировщик.

На рис. 4.18 показаны регистры контроллера ПДП, которые используются процессором для инициирования операции пересылки данных. Два регистра предназначены для хранения начального адреса и счетчика слов. В третьем содержатся информация о состоянии и управляющие флаги. Разряд $R/\overline{W}$ этого регистра определяет направление пересылки. Когда команда программы устанавливает этот разряд в 1, контроллер выполняет операцию чтения, то есть пересылает данные из памяти в устройство ввода-вывода. В противном случае он выполняет операцию записи. Контроллер, завершивший пересылку блока данных и готовый к обслуживанию следующей команды, устанавливает флаг DONE в 1. Разряд 30 соответствует флагу разрешения прерывания IE. Если этот флаг установлен в 1, то по окончании пересылки блока данных контроллер запрашивает прерывание, после чего устанавливает в 1 разряд IRQ.

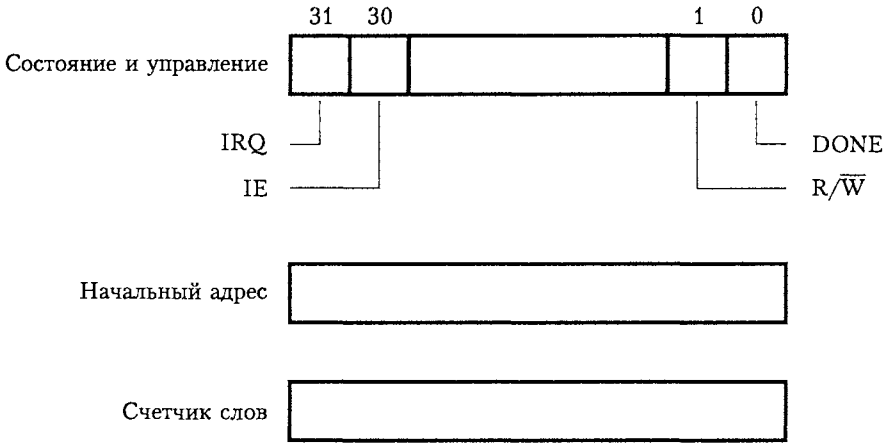


Рис. 4.18. Регистры интерфейса ПДП

Пример использования прямого доступа к памяти в компьютерной системе приведен на рис 4.19. Контроллер ПДП соединяет шину компьютера с высокоскоростной сетью. Контроллер, управляющий двумя дисками, тоже имеет встроенные функции ПДП и поддерживает два канала передачи данных. Он может выполнять две независимые операции прямого доступа к памяти так, словно каждый диск содержит собственный контроллер ПДП. Для этого в контроллере имеется два набора регистров, предназначенных для хранения адреса в памяти и счетчика слов, — по одному для каждого устройства.

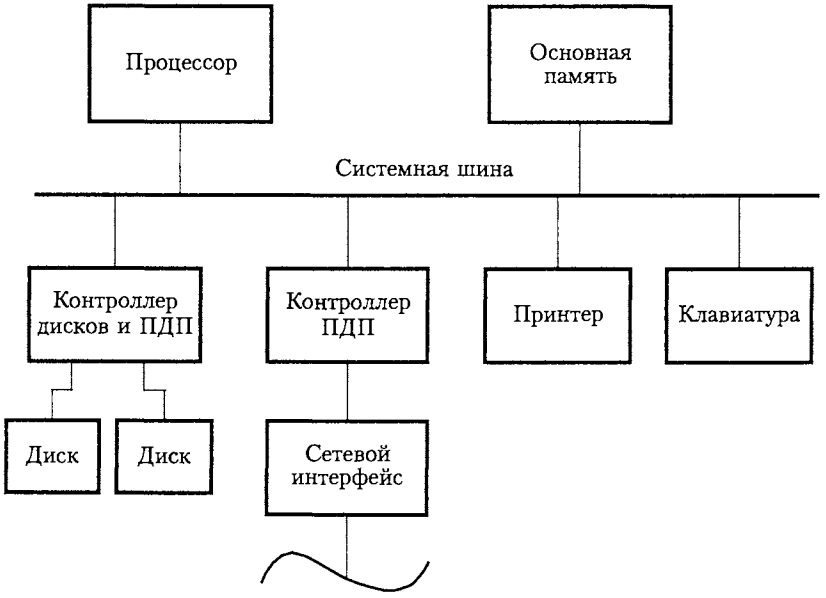


Рис. 4.19. Использование контролеров ПДП в компьютерной системе

Для того чтобы начать пересылку блока данных из основной памяти на один из дисков, программа записывает адрес и значение счетчика слов в регистры соответствующего канала ПДП дискового контроллера. Кроме того, она предоставляет контроллеру информацию, которая позволит идентифицировать данные в будущем, когда их потребуется прочитать с диска. После этого контроллер ПДП уже независимо от процессора выполняет указанную операцию. Когда она завершается, в регистре состояния и управления канала ПДП устанавливается разряд DONE. Если в это время разряд IE равен 1, контроллер направляет процессору запрос прерывания и устанавливает разряд IRQ. Регистр состояния может использоваться и для хранения другой информации, например значения индикатора, указывающего, успешно ли выполнена пересылка и не произошло ли при этом каких-либо ошибок.

Процессор и контроллеры ПДП обращаются к памяти поочередно. Запросы устройств ПДП на использование шины всегда имеют более высокий приоритет, чем запросы процессора. Среди устройств ПДП наивысшим приоритетом обладают высокоскоростные внешние устройства, в том числе диски, скоростной сетевой интерфейс, графический дисплей. Поскольку большая часть циклов доступа к памяти инициируется процессором, можно сказать, что контроллер ПДП «крадет» их у процессора. Применяемая при этом технология чередования называется *захватом циклов*. В качестве альтернативы контролеру ПДП для пересылки блока данных без прерываний может быть предоставлен монопольный доступ к основной памяти. Такой режим называется *блочным* (block mode) или *монопольным* (burst mode).

Большинство контроллеров ПДП содержат буфер для хранения данных. Например, в случае сетевого интерфейса (рис. 4.19) контролер ПДП считывает из основной памяти блок данных и сохраняет его в своем входном буфере. Пересылка выполняется в монопольном режиме при максимальной скорости, с которой могут работать память и шина компьютера. После этого данные пересылаются из буфера по сети со скоростью, определяемой пропускной способностью сетевого соединения.

Если процессор и контроллер ПДП или же два контроллера ПДП попытаются одновременно использовать шину для доступа к основной памяти, возникнет конфликт. Для разрешения конфликтных ситуаций применяется специальная процедура выбора, реализуемая схемами управления шиной и координирующая действия всех устройств, запрашивающих операции с памятью.

4.4.1. Разрешение конфликтов на шине

Устройство, которому в данный момент разрешается инициировать пересылку данных по шине, называется хозяином шины (bus master). Когда текущий хозяин шины перестает быть владельцем шины, эта роль может быть передана другому устройству. Выбор очередного устройства, которое станет хозяином шины, осуществляется посредством специальной процедуры, получившей название *арбитраж*. При этом учитываются потребности различных устройств, для чего опять-таки используется система приоритетов.

Существует два подхода к разрешению конфликтов на шине: централизованный арбитраж и распределенный арбитраж. При централизованном подходе разрешение конфликтов выполняется *арбитром шины*, а при распределенном подходе выбор следующего хозяина шины производится с участием всех устройств.

Централизованный арбитраж

Арбитром шины может служить как процессор, так и какое-либо отдельное устройство, подключенное к шине. На рис. 4.20 показана архитектура компьютера, в котором арбитражная схема входит в состав процессора. При такой архитектуре хозяином шины обычно является процессор, если только он не предоставит эту роль одному из контроллеров ПДП. Когда контроллеру ПДП требуется шина, он активизирует линию запроса шины $\overline{BR}$ (Bus-Request). Эта линия подобна линии запроса прерывания (рис. 4.6). Сигнал на линии запроса шины является логической суммой (ИЛИ) сигналов от всех подсоединенных к данной линии устройств. Когда линия запроса шины активизируется, процессор передает сигнал предоставления шины BG1 (Bus-Grant), указывающий контроллерам ПДП, что как только шина освободится, они смогут ее использовать. Линия, по которой передается этот сигнал, соединяется со всеми контроллерами ПДП по принципу гирляндной цепи. Таким образом, если контроллер ПДП 1 запросит управление шиной, он заблокирует передачу сигнала ее предоставления остальным устройствам. Если же шина ему не нужна, он передаст этот сигнал дальше, то есть на его выход будет подан сигнал BG2. Текущий хозяин шины указывает прочим устройствам, что шина занята, с помощью сигнала $\overline{BBSY}$. Получив сигнал предоставления шины, контроллер ПДП ждет, когда будет снят сигнал $\overline{BBSY}$, и только после этого назначает хозяином шины новое устройство. Он тут же активизирует сигнал о том, что шина занята, чтобы никакие другие устройства не могли использовать таковую одновременно с ее новым хозяином.

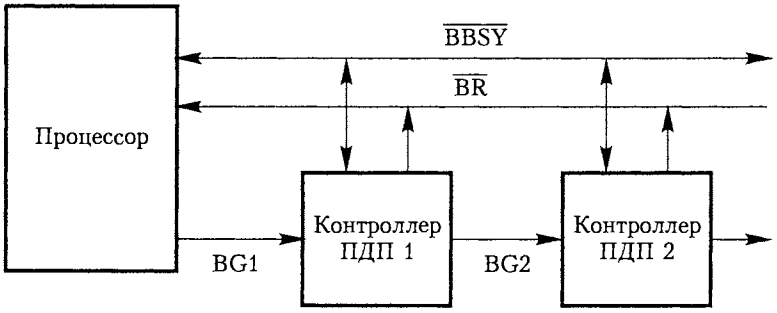


Рис. 4.20. Простая архитектура управления, построенная по принципу гирляндной цепи

Временная диаграмма, приведенная на рис. 4.21, отражает последовательность событий в описанной выше сетевой архитектуре, происходящую с того момента, как контроллер ПДП 2 запрашивает управление шиной, которую он впоследствии должен освободить. Пока контроллер является хозяином шины, он может выполнить одну или более операций пересылки данных в зависимости от того,

работает он в блочном режиме или в режиме захвата шины. Как только контроллер освобождает шину, ее хозяином снова становится процессор. На рис. 4.21 показана причинно-следственная связь между сигналами, участвующими в арбитражном процессе. Особенности тактирования, зависящие от модели конкретного компьютера, на этом рисунке не отражены.

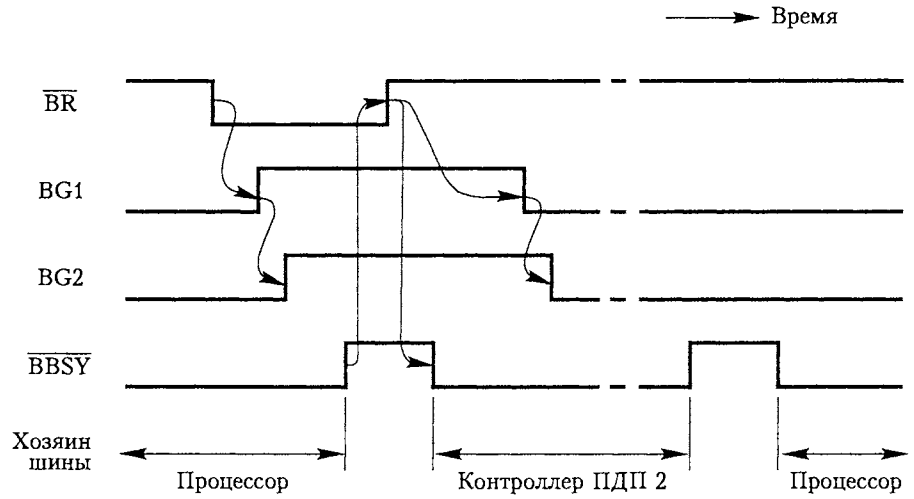


Рис. 4.21. Последовательность сигналов при передаче управления шиной от одного устройства к другому для архитектуры, представленной на рис. 4.20

На рис. 4.20 показаны одна линия запроса и одна линия предоставления шины. Но вообще-то в компьютере таких пар линий может быть несколько, равно как и линий запроса прерываний (рис. 4.8, *б*). Подобная архитектура отличается значительно большей гибкостью с точки зрения определения порядка обслуживания устройств. Арбитражная схема гарантирует поочередное обслуживание запросов в соответствии с приоритетами устройств. Например, если имеется четыре линии запроса шины, от BR1 до BR4, то может применяться фиксированная схема приоритетов, согласно которой линия BR1 имеет самый высокий приоритет, а линия BR4 — самый низкий. В качестве альтернативы может быть использована схема с циклическим чередованием приоритетов, при которой все устройства имеют равные шансы на обслуживание. Циклическое чередование приоритетов означает, что после обслуживания запроса по линии BR1 их порядок становится следующим: 2, 3, 4, 1.

Распределенный арбитраж

При *распределенном арбитраже* все устройства, ожидающие своей очереди на использование шины, на равных правах участвуют в арбитражном процессе, так как централизованного арбитра в такой схеме не существует. Простейший способ распределенного арбитража показан на рис. 4.22. Каждому соединенному с шиной устройству назначается 4-разрядный идентификационный номер. Когда одно или несколько устройств запрашивают шину, они активизируют сигнал начала

арбитража $\overline{\text{Start-Arbitration}}$ и помещают свои 4-разрядные идентификационные номера на линии с открытым коллектором от $\overline{\text{ARB0}}$ до $\overline{\text{ARB3}}$. Победитель определяется в результате обработки сигналов, переданных по линиям всеми претендентами. Выходом этой сети является код на четырех линиях, представляющий запрос с наивысшим идентификационным номером.

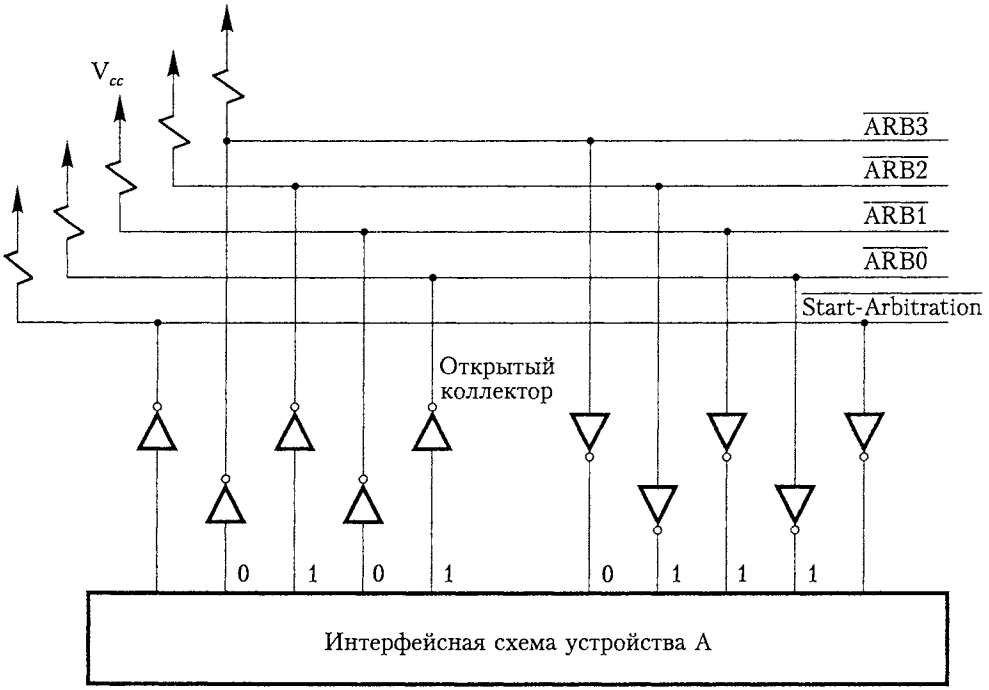


Рис. 4.22. Схема распределенного арбитража

В описанной схеме используются повторители с открытым коллектором. Поэтому, если значением на входе одного повторителя является 1, а значением на входе другого повторителя, соединенного с той же линией шины, — 0, то на линии будет низкое напряжение. Иными словами, такое соединение выполняет функцию ИЛИ, в которой выигрывает логическое значение 1.

Предположим, что устройства А и В, которые имеют идентификационные номера 5 и 6, одновременно запрашивают шину. Устройство А передает значение 0101, а устройство В — значение 0110. Оба они получают код 0111. После этого каждое из устройств сравнивает код со своим идентификационным номером, начиная с самого старшего разряда, и, если обнаруживает различие хотя бы в одном разряде, отключает свой повторитель в этом разряде и все повторители младших разрядов. Для этого оно просто помещает на входы соответствующих повторителей значение 0. В нашем примере устройство А обнаруживает различие на линии $\overline{\text{ARB1}}$, поэтому отключает повторители на линиях $\overline{\text{ARB1}}$ и $\overline{\text{ARB0}}$. В результате код

на линиях арбитража меняется на 0110, и это означает, что победило устройство B. Обратите внимание, что поскольку код на линиях приоритета в течение короткого промежутка времени был равен 0111, устройство B смогло временно отключить свой повторитель на линии $\overline{\text{ARB0}}$. Однако оно снова включит этот повторитель, как только увидит на линии $\overline{\text{ARB1}}$ значение 0, появившееся в результате действий устройства A.

Децентрализованный арбитраж считается более надежным, поскольку функционирование шины не зависит от одного-единственного устройства. Он реализован во многих схемах, и в частности в архитектуре шины SCSI, о которой рассказывается в разделе 4.7.2.

4.5. Шины

Процессор, основная память и устройства ввода-вывода могут соединяться между собой посредством общей шины, основным назначением которой является предоставление канала связи для пересылки данных.

Шина содержит линии для поддержки прерываний и арбитража. В этом разделе мы обсудим основные особенности шинных протоколов, используемых для пересылки данных. Шинный протокол — это набор правил, управляющих поведением соединенных с шиной устройств, а также последовательностью помещения информации на шину, выдачи управляющих сигналов и т. п. После обсуждения шинных протоколов мы рассмотрим примеры интерфейсных схем, в которых они используются.

Линии шины, используемые для пересылки данных, бывают трех типов: линии данных, линии адреса и управляющие линии. Управляющие сигналы определяют, какую операцию, чтения или записи, следует выполнить. Обычно для этой цели используется линия $\text{R}/\overline{\text{W}}$. Значение 1 на этой линии соответствует операции чтения, а значение 0 — операции записи. Когда команда допускает использование операндов разных размеров, например, байтов, слов и длинных слов, размер данных также указывается на управляющих линиях.

Сигналы управления шиной также используются для тактирования операций. Они определяют, в какой момент процессор и устройства ввода-вывода могут поместить данные на шину или прочитать их с таковой. Для тактирования пересылки данных по шине разработано множество схем, которые можно разделить на два основных типа: *синхронные* и *асинхронные*.

Как уже было отмечено выше, в любой операции пересылки данных по шине одно из устройств играет роль хозяина шины. Это устройство инициирует пересылку данных с помощью команд чтения или записи. Поэтому его можно назвать *инициатором*. Обычно хозяином шины является процессор, но эту роль могут выполнять и другие устройства, поддерживающие функцию прямого доступа к памяти. Устройство, к которому обращается хозяин шины, называется *подчиненным* или *целевым*.

4.5.1. Синхронная шина

В случае *синхронной* шины все устройства получают синхронизирующую информацию по общей тактовой линии. На эту линию подаются тактовые импульсы со строго фиксированной частотой. Промежуток времени между последовательными тактовыми импульсами в простейшей синхронной шине составляет *цикл шины*, в течение которого выполняется одна операция пересылки данных. Такая схема показана на рис. 4.23. На этом и следующих рисунках на линиях адреса и данных показаны сигналы и низкого и высокого уровня. Это стандартный прием, обозначающий, что на одних из этих линий уровень сигнала низкий, а на других, наоборот, высокий и что все зависит от конкретных значений адреса и данных. Точки пересечения линий на рисунке соответствуют моментам изменения этих значений. Сигнальная линия в неопределенном, или высокоимпедансном, состоянии представлена промежуточным уровнем, находящимся посередине между высоким и низким.

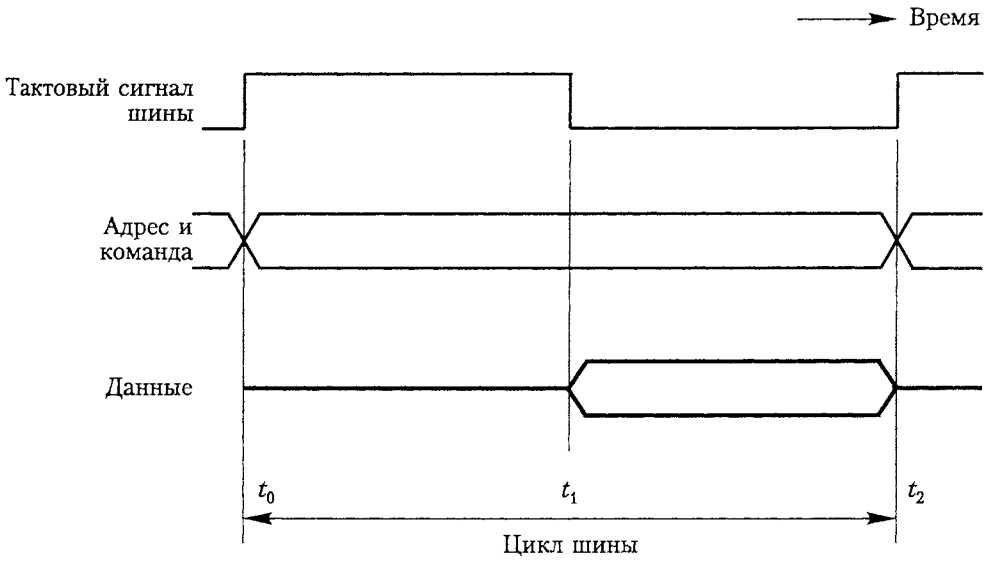


Рис. 4.23. Временная диаграмма операции пересылки по синхронной шине при вводе данных

Далее будет рассмотрена последовательность событий, происходящих при выполнении операции ввода (чтения) данных. В момент времени t_0 хозяин шины помещает на адресные линии адрес устройства и отсылает по управляющим линиям необходимую команду. В этой команде определяется операция ввода и, если нужно, задается длина считываемого операнда. Информация передается по шине со скоростью, определяемой ее физическими и электрическими характеристиками. Длительность тактового импульса $t_1 - t_0$ должна быть больше максимального времени задержки на распространение сигнала между двумя соединенными с шиной устройствами. Причем она должна быть достаточно большой, так как все устройства

должны успеть декодировать адрес и управляющие сигналы, с тем чтобы адресуемое (подчиненное) устройство могло ответить на команду в момент времени t_1. В течение промежутка времени от t_0 до t_1 информация на шине ненадежна, поскольку состояние сигналов изменяется. В момент времени t_1 адресуемое подчиненное устройство помещает запрошенные входные данные на линии данных.

В конце тактового цикла, то есть в момент времени t_2, хозяин шины *стробирует* данные на линиях данных в свой входной буфер. Слово «стробировать» в используемом контексте означает снять значения с линий данных в указанный момент времени и сохранить их в буфере. Для того чтобы данные правильно загружались в любое устройство хранения, в том числе в регистр на основе триггеров, они должны находиться на его входе в течение времени, достаточного для их сохранения (см. приложение А). Поэтому период времени $t_2 - t_1$ должен быть больше максимального времени распространения сигнала по шине в сумме со временем установки входного буферного регистра хозяина шины.

Похожая процедура выполняется и при выводе данных. Хозяин шины помещает на линии данных выходные сведения, а на линии адреса и управляющие линии — адрес и команду. В момент времени t_2 адресуемое устройство стробирует линии данных и загружает информацию в свой буфер данных.

На временной диаграмме, приведенной на рис. 4.23, происходящее на линиях шины показано в очень упрощенном виде. На самом деле из-за задержек на распространение сигналов по проводам шины и в схемах устройств точные моменты изменения сигналов несколько не соответствуют указанным на рисунке. Временная диаграмма на рис. 4.24 более реалистична. Каждый сигнал, за исключением тактового, показан здесь в двух вариантах. Поскольку на передачу сигнала от одного устройства к другому уходит некоторое время, разные устройства видят изменения этого сигнала в разные моменты. Одно представление соответствует тому, как данный сигнал видит хозяин шины, а другое — тому, как его видит подчиненное устройство. При этом предполагается, что изменения тактового сигнала все подключенные к шине устройства замечают одновременно. Конструкторами систем прилагаются огромные усилия для того, чтобы тактовый сигнал удовлетворял этому требованию.

В начале такта 1 (t_0), на переднем фронте тактового сигнала, хозяин шины передает по ней сигналы адреса и команды. Однако до момента времени t_{AM} эти сигналы не появляются на шине, прежде всего из-за задержки в схеме управления шиной. Какое-то время спустя, в момент t_{AS}, сигналы достигают подчиненного устройства. Последнее декодирует адрес и в момент времени t_1 отсылает запрошенные данные. И опять эти сигналы не появляются на шине сразу, а задерживаются до момента t_{DS}. К хозяину шины они прибывают в момент времени t_{DM}. Далее, в момент времени t_2, хозяин загружает данные в свой входной буфер. Промежуток $t_2 - t_{DM}$ является временем установки его входного буфера. После момента времени t_2 данные должны оставаться неизменными в течение всего периода их хранения в буфере.

В литературе обычно приводятся упрощенные временные диаграммы, как на рис. 4.23, дающие концептуальное представление о процессе пересылки данных. Но реальные сигналы всегда распространяются с задержками (рис. 4.24).

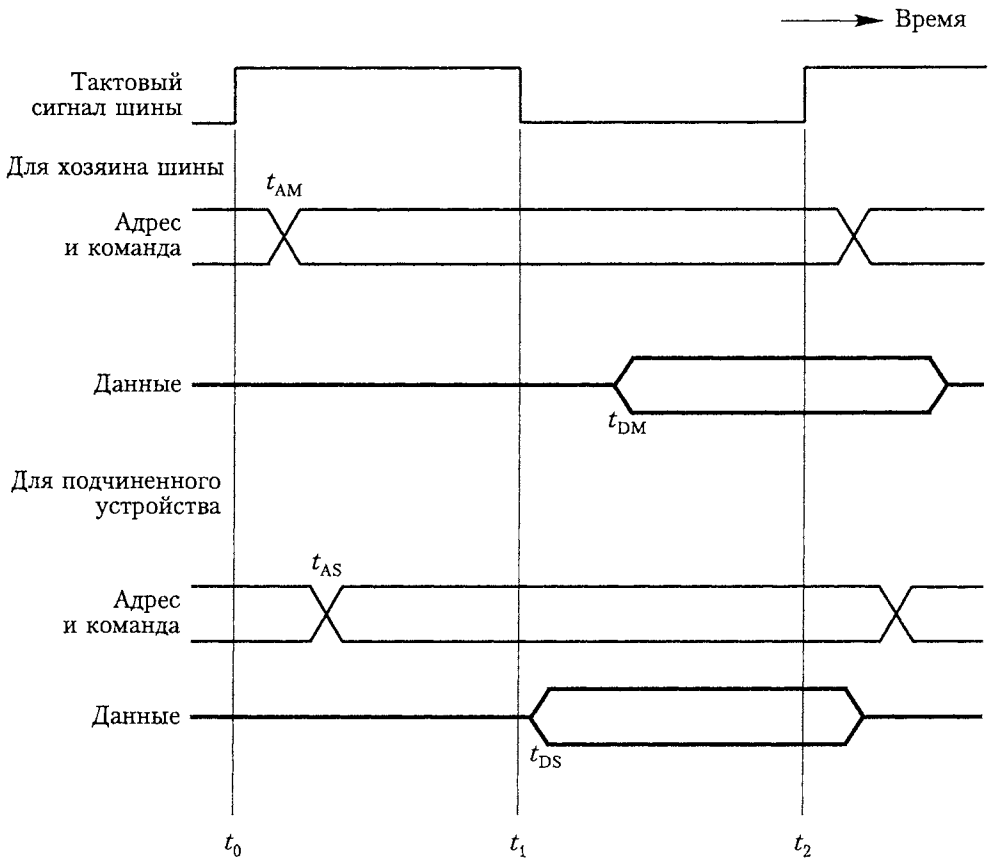


Рис. 4.24. Подробная временная диаграмма для операции пересылки входных данных, показанной на рис. 4.23

Пересылка данных за несколько тактов

Описанная выше схема пересылки данных упрощает разработку интерфейса устройств, но она имеет ряд ограничений. Поскольку пересылка данных, согласно этой схеме, выполняется за один такт, период $t_2 - t_0$ должен выбираться таким образом, чтобы вместить и наиболее долгие задержки на шине, и задержки самого медленного интерфейса устройств. В результате все устройства будут работать со скоростью самого медленного из них.

Кроме того, у процессора нет возможности определить, какое из адресуемых устройств на самом деле ответило на запрос. Он просто предполагает, что в момент времени t_2 выходные данные получены устройством ввода-вывода или что входные данные имеются на линиях данных. Если же по какой-либо причине устройство не ответит, процессор даже не обнаружит ошибку.

Для преодоления этих ограничений в большинство шин включают поддержку управляющих сигналов, передаваемых в качестве ответа устройства. Эти сигналы

информируют хозяина шины о том, что подчиненное устройство распознало адрес и готово участвовать в операции пересылки данных. Кроме того, они позволяют откорректировать длительность периода пересылки данных в соответствии с требованиями участвующих в операции устройств. Для упрощения этой задачи используется тактовый сигнал высокой частоты, при котором цикл пересылки занимает несколько тактов. Таким образом, количество тактов, затрачиваемых на операцию пересылки данных, зависит от конкретной пары устройств.

Пример реализации такого подхода приведен на рис. 4.25. В течение такта 1 хозяин шины пересылает адрес и информацию о команде, запрашивая операцию чтения. Подчиненное устройство получает и декодирует эту информацию. По следующему переднему фронту тактового сигнала, то есть в начале такта 2, устройство принимает решение ответить на запрос и начинает процедуру доступа к запрошенным данным. Как известно, на извлечение данных требуется некоторое время, поэтому подчиненное устройство не может ответить немедленно. На третьем такте данные готовы и помещаются на шину. В то же время подчиненное устройство выдает управляющий сигнал, называемый Slave-ready (подчиненный готов). Хозяин шины, ожидавший этого сигнала, стробирует данные в свой входной буфер в конце такта 3. На этом операция пересылки данных по шине заканчивается, и хозяин шины может отправить по ней новый адрес, чтобы на такте 4 начать другую операцию пересылки.

Сигнал Slave-ready — это направляемое подчиненным устройством хозяину шины сообщение, говорящее о том, что им отправлены правильные данные. В нашем примере подчиненное устройство отвечает во время такта 3. Другое устройство может ответить раньше или позже. Таким образом, сигнал Slave-ready позволяет изменять длительность операции пересылки данных в соответствии с возможностями конкретного устройства. Если адресуемое устройство не отвечает, хозяин шины ждет в течение заранее определенного количества тактов, а затем отменяет операцию. Отсутствие ответа может быть результатом отправки неверного адреса или некорректного функционирования устройства.

Обратите внимание, что для тактирования шины не всегда используется тот же сигнал, что и для тактирования процессора. Последний обычно имеет значительно более высокую частоту, поскольку управляет внутренним функционированием микросхемы процессора. Задержки на распространение сигналов внутри процессора гораздо меньше, чем задержки на шине, связывающей, в частности, микросхемы на печатных платах. То, какая тактовая частота может использоваться в конкретном компьютере, зависит от технологии его производства. Для современных процессорных микросхем типична тактовая частота свыше 500 МГц, для памяти и шин ввода-вывода может использоваться частота от 50 до 150 МГц.

Многие компьютерные шины, в том числе шины процессоров Pentium и ARM, основаны на схемах, подобных показанной на рис. 4.25. Очень похож на эту схему и стандарт архитектуры шины PCI, описанный в разделе 4.7.1. Далее мы рассмотрим иной подход к созданию шин, когда тактовый сигнал вообще не используется.

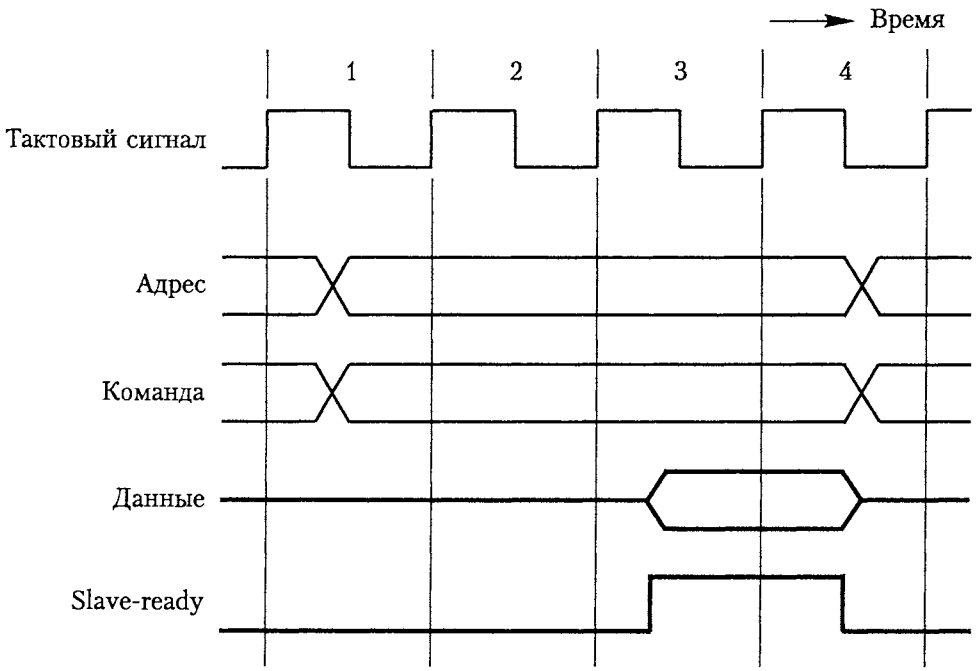


Рис. 4.25. Пересылка данных с использованием нескольких тактов

4.5.2. Асинхронные шины

Альтернативная схема управления пересылкой данных по шине основывается на механизме *квитирования*, то есть подтверждения связи, между хозяином шины и подчиненным устройством. Концепция квитирования является обобщением идеи использования сигнала Slave-ready, проиллюстрированной на рис. 4.25. В схеме с квитированием тактовая линия заменяется двумя управляющими линиями синхронизации: Master-ready и Slave-ready. Первая принадлежит хозяину шины, который передает по ней сигнал готовности к транзакции, а по второй отвечает подчиненное устройство.

Пересылка данных, управление которой осуществляется посредством протокола с квитированием, выполняется следующим образом. Хозяин шины помещает на нее адрес и информацию о команде. Затем по линии Master-ready он сообщает об этом всем устройствам. В ответ подключенные к шине устройства декодируют адрес. То устройство, для которого предназначена команда, выполняет таковую и информирует об этом хозяина шины по линии Slave-ready. Хозяин дожидается этого сигнала и только после этого удаляет с шины свои сигналы. В случае операции чтения он стробирует данные в свой входной буфер.

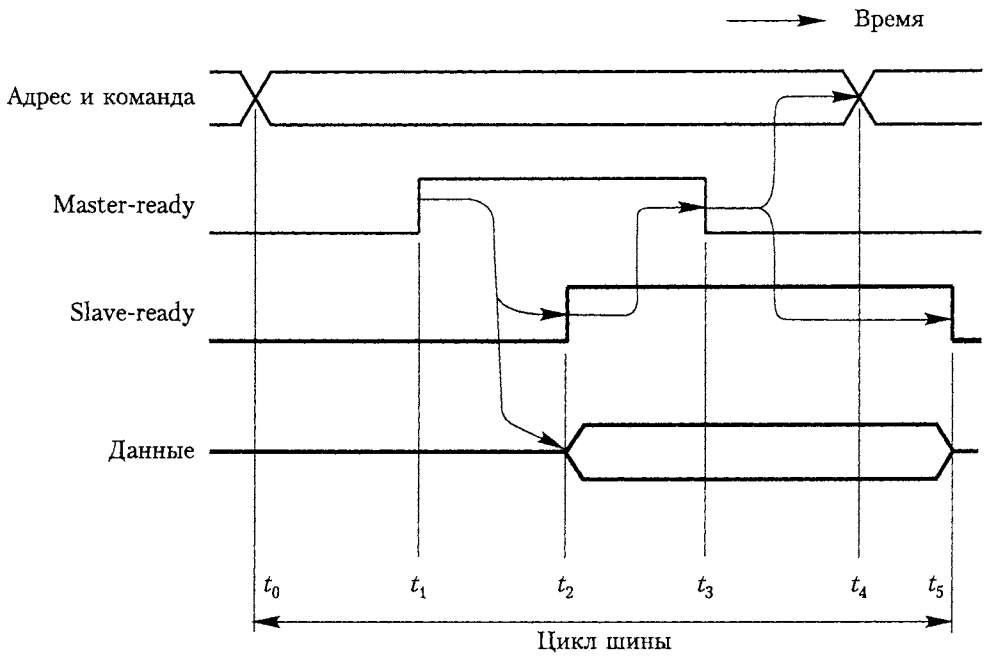


Рис. 4.26. Пересылка входных данных по шине с использованием квитирования

Пример выполнения операции пересылки входных данных с использованием квитирования проиллюстрирован на рис. 4.26, отражающем следующую последовательность событий.

t_0 — хозяин шины помещает на нее адрес и команду, и все устройства на шине начинают декодировать эту информацию.

t_1 — хозяин шины активизирует линию Master-ready, чтобы проинформировать устройства ввода-вывода о том, что адрес и команда поданы на шину. Задержка, равная $t_1 - t_0$, формируется на тот случай, если на шине произойдет *сдвиг* сигналов. Сдвигом называется неодновременное прибытие в точку назначения двух сигналов, одновременно выданных источником. Причиной такого сдвига может стать разное время распространения сигнала по различным линиям шины. Для того чтобы иметь гарантию, что сигнал Master-ready не достигнет какого-либо устройства раньше адреса и команды, нужно задержать его на время $t_1 - t_0$, которое больше времени максимально возможного сдвига. (В случае синхронной шины сдвиг на шине тоже учитывается — он является составной частью максимальной задержки на распространение сигнала.) Когда адресная информация достигает очередного устройства, она декодируется его интерфейсной схемой. На декодирование тоже уходит какое-то время, которое следует включить в период $t_1 - t_0$.

t_2 — после декодирования адреса и команды подчиненное устройство выполняет операцию ввода, то есть перемещает информацию из своего регистра данных

на линии данных. Одновременно оно активизирует сигнал Slave-ready. Если интерфейсная схема помещает данные на шину с некоторой задержкой, подчиненное устройство должно соответственно задержать и выдачу сигнала Slave-ready. Длительность промежутка времени $t_2 - t_1$ зависит от расстояния между хозяином шины и подчиненным устройством, а также от задержек в схеме подчиненного устройства. Именно потому, что указанный промежуток времени имеет разную длительность, схема получается асинхронной.

t_3 — сигнал Slave-ready достигает хозяина шины и сообщает ему, что на шине имеются данные. Однако поскольку предполагается, что интерфейс устройства помещает на шину сигнал Slave-ready одновременно с данными, хозяин шины должен выдержать небольшую паузу на случай сдвига сигналов. Кроме того, он должен учесть время установки собственного входного буфера. После задержки, равной сумме времени максимального сдвига сигналов на шине и минимального времени установки буфера, хозяин шины стробирует данные в свой входной буфер. Одновременно он удаляет с шины сигнал Master-ready, сообщая тем самым о получении данных.

t_4 — хозяин шины удаляет с шины адрес и команду. Задержка между моментами времени t_3 и t_4 производится на случай сдвига на шине. Если адрес, который видит какое-либо из устройств, начнет изменяться, пока сигнал Master-ready равен 1, адресация будет выполнена неточно.

t_5 — когда интерфейс устройства фиксирует переход сигнала Master-ready из 1 в 0, он удаляет с шины данные и сигнал Slave-ready. На этом пересылка входных данных завершается.

Процесс выполнения операции вывода (рис. 4.27) почти ничем не отличается от процесса выполнения операции ввода. Правда, в первом случае хозяин шины одновременно с адресом и командой помещает на линии данных выходную информацию. Получив сигнал Master-ready, подчиненное устройство стробирует данные в свой входной буфер и сообщает об этом установкой сигнала Slave-ready в 1. Дальше все происходит точно так, как при вводе данных.

При построении временных диаграмм, приведенных на рис. 4.26 и 4.27, предполагалось, что хозяин шины компенсирует сдвиг на шине и задержку на декодирование адреса. Для этой цели предназначены задержки $t_1 - t_0$ и $t_4 - t_3$. Если длительность задержки достаточна для декодирования адреса интерфейсом устройства ввода-вывода, интерфейсная схема может использовать сигнал Master-ready для пропуска других сигналов на шину и с шины. Сказанное станет понятнее, когда мы рассмотрим примеры интерфейсных схем, приведенные далее в этой главе.

Сигналы квитирования в приведенных нами схемах взаимосвязаны таким образом, что за изменением одного сигнала следует изменение другого. Поэтому такая схема, называемая *полным квитированием*, обладает исключительной гибкостью и надежностью.

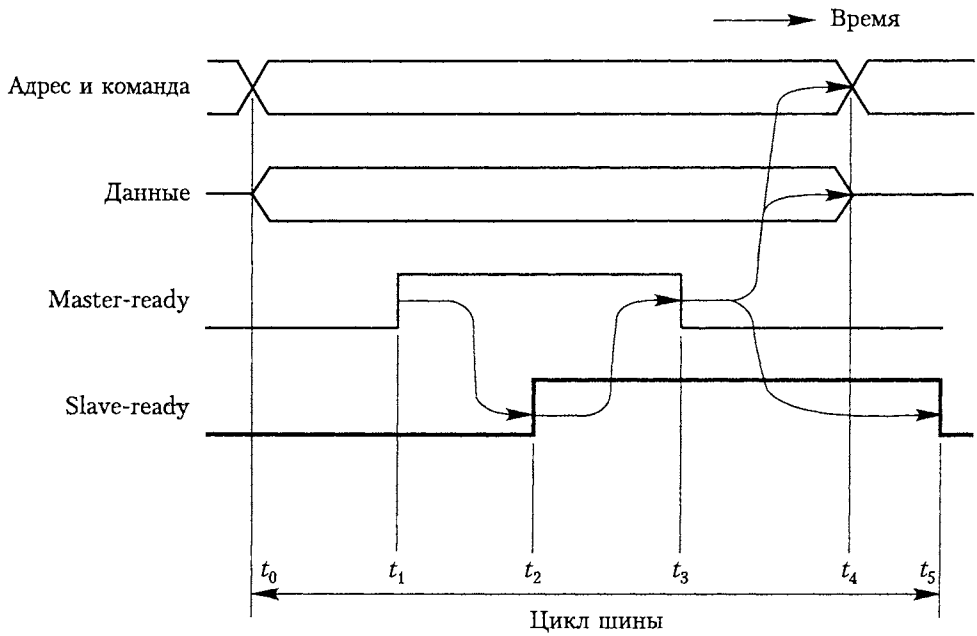


Рис. 4.27. Пересылка выходных данных по шине с использованием квитирования

4.5.3. Заключительные замечания

В современных компьютерах реализовано множество разновидностей рассмотренных нами технологий. Например, шина семейства процессоров 68000 может работать в двух режимах: синхронном и асинхронном. Выбирая архитектуру шины, следует учитывать такие факторы:

- простоту интерфейса устройства;
- способность работать с интерфейсами устройств, имеющими разное время задержки;
- общее время, уходящее на пересылку данных по шине;
- способность обнаруживать ошибки, происходящие в результате адресации несуществующего устройства или неверного функционирования интерфейса устройства.

Важнейшим преимуществом асинхронной шины является то, что процесс квитирования избавляет конструктора от необходимости синхронизации тактовых сигналов отправителя и получателя, что значительно упрощает разработку. Никакие задержки, связанные с распространением сигнала по шине или с интерфейсными схемами, не отражаются на работе системы. Когда величина таких задержек изменяется, например, из-за изменения нагрузки при добавлении или удалении интерфейсной схемы, автоматически изменяется и время передачи данных. Для синхронных шин схемы тактирования должны разрабатываться очень тщательно, что обеспечит правильную синхронизацию, а задержки не должны превышать строго рассчитанный предел.

Скорость передачи данных по асинхронной шине, управляемой посредством механизма полного квитирования, несколько снижается из-за того, что каждая такая операция выполняется с четырьмя задержками, по две в каждую сторону. Это хорошо видно на рис. 4.26 и 4.27: каждое изменение сигнала на линии Master-ready происходит только после изменения сигнала на линии Slave-ready и наоборот. В случае синхронной шины тактовый период должен включать задержку на распространение сигнала только в одном направлении. Благодаря этому пересылка выполняется быстрее. Для более медленных устройств, как уже было сказано, используются дополнительные такты. Поэтому высокоскоростные шины большинства современных компьютеров имеют синхронную архитектуру.

4.6. Интерфейсные схемы

Интерфейс устройства ввода-вывода представляет собой схему, соединяющую устройство с шиной компьютера. По одну сторону этой схемы расположены сигналы шины для адреса и данных, а также управляющие сигналы, по другую сторону, называемую *портом*, — линии для передачи данных и управляющих сигналов между интерфейсом и устройством ввода-вывода. Порт может быть параллельным или последовательным. Параллельный порт одновременно пересылает от устройства или к устройству группу битов данных, обычно 8 или 16. Последовательный порт пересылает данные по одному биту за раз. Взаимодействие с шиной в обоих случаях осуществляется по одному принципу, а взаимопреобразование последовательного и параллельного форматов выполняется внутри интерфейсной схемы.

В случае параллельного порта для соединения между устройством и компьютером используется многоконтактный разъем и кабель с соответствующим количеством проводов, как правило, плоский. Схемы на обоих концах относительно просты, поскольку им не нужно выполнять взаимопреобразование последовательного и параллельного форматов данных. Это удобно для устройств, которые физически располагаются близко от компьютера. Если же расстояние от компьютера до устройства достаточно велико, скорость передачи данных резко снижается из-за сдвига сигнала, о котором упоминалось в предыдущем разделе. Поэтому, если требуется длинный кабель, гораздо удобнее и дешевле использовать последовательный формат. О последовательных форматах передачи данных мы подробно поговорим в главе 10.

Прежде чем перейти к обсуждению примера интерфейсной схемы, давайте еще раз перечислим функции интерфейса ввода-вывода. Согласно разделу 4.1, интерфейс ввода-вывода выполняет следующие функции.

- предоставляет буфер для хранения как минимум одного слова данных (или одного байта, как в случае байт-ориентированных устройств);
- содержит доступные процессору флаги состояния, по которым тот может определить, заполнен буфер (в случае ввода данных) или он пуст (в случае вывода);
- содержит схему декодирования адреса, позволяющую устройству определить, когда оно адресуется процессором;

- генерирует тактовые сигналы, необходимые для функционирования схемы управления шиной;
- выполняет преобразование формата, если таковое требуется для пересылки данных между шиной и устройством ввода-вывода (например, преобразование из параллельного формата в последовательный, необходимое для отправки данных через последовательный порт).

4.6.1. Параллельный порт

В данном разделе важнейшие аспекты архитектуры интерфейсных схем будут рассмотрены на конкретных примерах. Сначала мы обсудим схемы для 8-разрядных входного и выходного портов, затем, объединив эти две схемы, покажем, какой должна быть структура интерфейса 8-разрядного параллельного порта общего назначения. При этом будет сделано предположение, что интерфейсная схема соединена с 32-разрядным процессором, использующим ввод-вывод с отображением в памяти и асинхронный шинный протокол, принцип действия которого показан на рис. 4.26 и 4.27. Кроме того, мы покажем, как модифицировать полученную схему для синхронного шинного протокола, представленного на рис. 4.25.

О том, какие аппаратные компоненты необходимы для соединения клавиатуры с процессором, можно судить по рис. 4.28. Типичная клавиатура содержит механические ключи, которые в нормальном состоянии открыты. Когда нажимается одна из клавиш, ее ключ замыкается, создавая соединение для прохождения электрического сигнала. Этот сигнал обнаруживается кодирующей схемой, которая генерирует ASCII-код соответствующего символа. Недостатком кнопочных ключей является то, что при нажатии кнопки контакт дребезжит. И хотя такое дребезжание обычно длится 1–2 мс, этого достаточно, чтобы компьютер воспринял одно нажатие клавиши как несколько последовательных электрических событий. Другими словами, одно нажатие клавиши может быть ошибочно интерпретировано компьютером как несколько быстрых нажатий и отпусканий. Эффект дребезжания можно устранить одним из двух способов: с помощью простой специальной схемы или программным путем. Когда применяется второй способ, программа ввода-вывода, считывающая символ с клавиатуры, просто ждет, пока контакт не прекратит дребезжать. Аппаратное решение — входящая в состав блока кодирования схема, способная устранить данное явление, — показано на рис. 4.28.

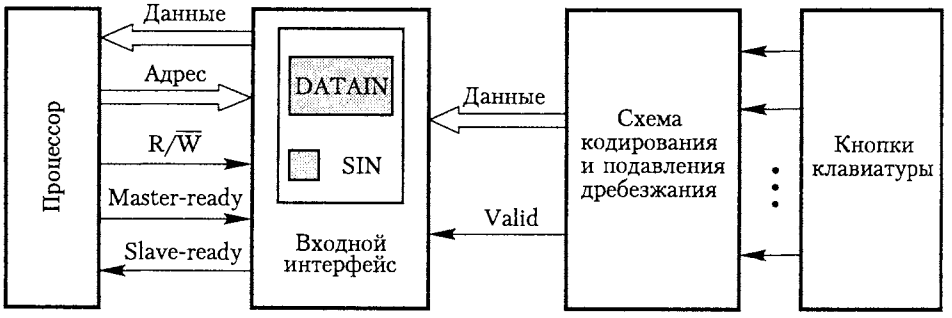


Рис. 4.28. Схема соединения клавиатуры с процессором

Выходной сигнал схемы кодирования состоит из набора битов, представляющего закодированный символ, и одного управляющего сигнала, называемого Valid и указывающего, что клавиша нажата. Эта информация направляется интерфейсной схеме, которая содержит регистр данных DATAIN и флаг состояния SIN. При нажатии клавиши сигнал Valid изменяется с 0 на 1, в результате чего ASCII-код символа загружается в регистр DATAIN, а флаг SIN устанавливается в 1. После того как процессор прочитает содержимое регистра DATAIN, этот флаг будет очищен. Интерфейсная схема соединена с асинхронной шиной, по которой данные пересылаются в процессор с использованием сигналов Master-ready и Slave-ready (рис. 4.26). Третья управляющая линия, R/$\overline{W}$, предназначена для различения операций чтения и записи. Рассмотрим интерфейсную схему (рис. 4.29).

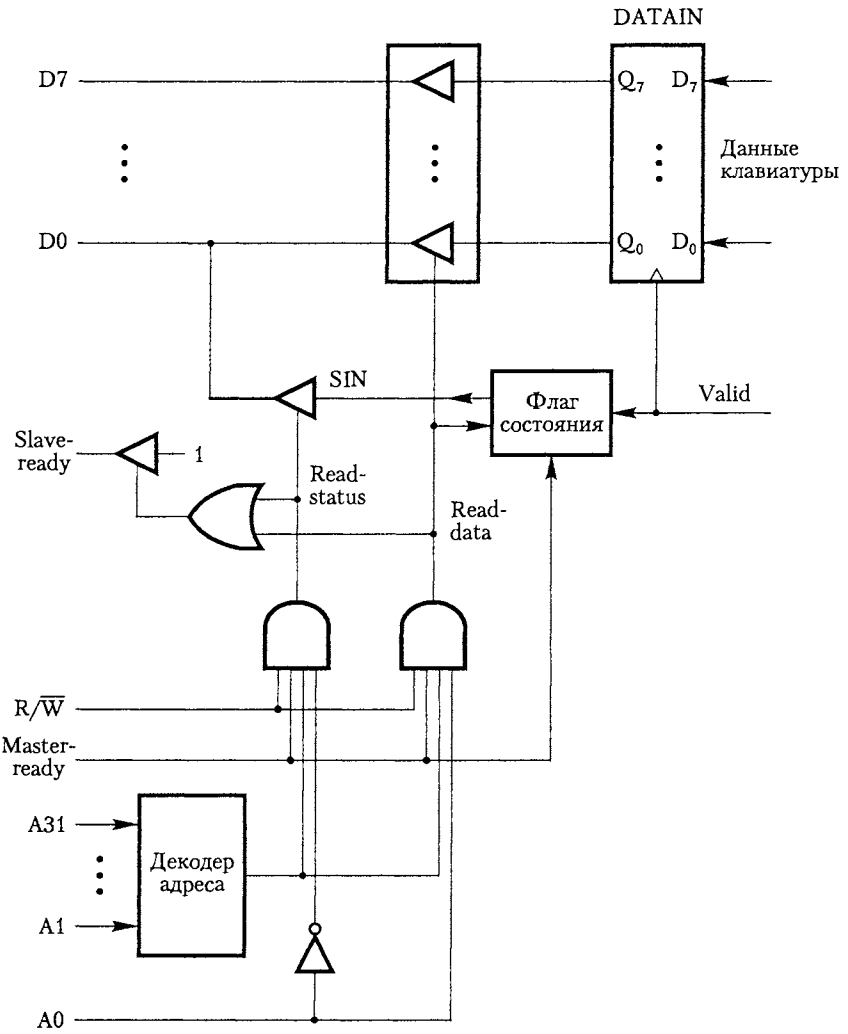


Рис. 4.29. Интерфейсная схема ввода

Выходные линии регистра DATAIN соединяются с линиями шины данных при помощи повторителей с тремя состояниями, которые включаются, когда процессор генерирует команду чтения с адресом этого регистра. Сигнал SIN генерируется схемой флага состояния. Данный сигнал тоже подается на шину через повторитель с тремя состояниями. Он соединяется с битом D0 и выглядит как нулевой разряд регистра состояния. Другие разряды этого регистра не содержат полезной информации. Декодер адреса предназначен для выбора интерфейса устройства ввода. Адрес используется в том случае, если один из назначенных интерфейсу адресов совпадает со старшими 31 разрядами адреса на шине. Адресный разряд A0 определяет, какой из регистров следует прочитать в ответ на активизацию сигнала Master-ready: регистр данных или регистр состояния. Аналогичным образом активизируются сигналы Read-data и Read-status. Квитирование выполняется путем активизации сигнала Slave-ready, когда значение Read-data или Read-status равно 1.

На рис. 4.30 представлен один из возможных вариантов реализации схемы флага состояния. Управляемый фронтом сигнала D-триггер устанавливается в 1 передним фронтом сигнала на линии Valid. Это событие изменяет состояние защелки ИЛИ-НЕ таким образом, что значение флага SIN становится равным 1. Пока флаг SIN считывается процессором, состояние этой защелки не изменяется. Флаг SIN может быть установлен только в том случае, если сигнал Master-ready равен 0. Когда сигнал Read-data активизируется для чтения регистра DATAIN, значения триггера и защелки сбрасываются в 0.

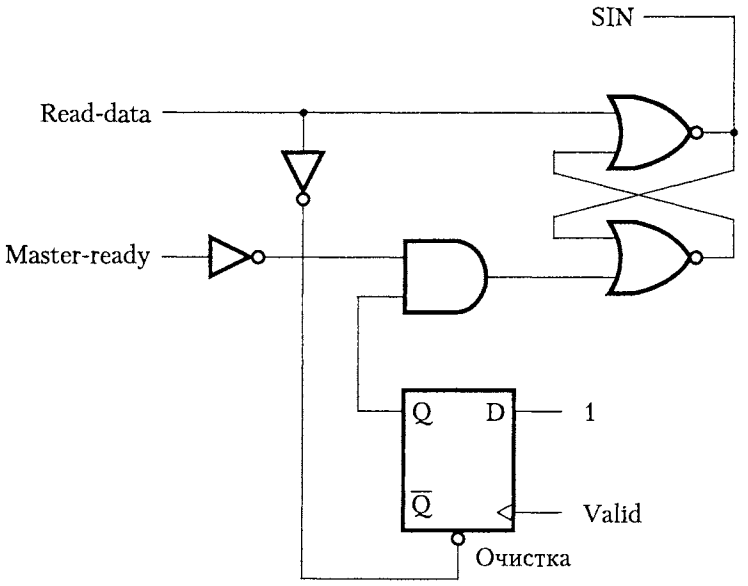


Рис. 4.30. Схема блока формирования флага состояния, показанного на рис. 4.29

Теперь давайте рассмотрим интерфейс выходного устройства, который может использоваться для подключения, скажем, принтера (рис. 4.31). Принтер работает

под управлением сигналов квитирования Valid и Idle, роль которых подобна роли сигналов Master-ready и Slave-ready. Когда принтер готов принять символ, он помещает на шину сигнал Idle. После этого интерфейсная схема может поместить на линии данных новый символ и активизировать сигнал Valid. В ответ принтер начинает печатать новый символ и удаляет сигнал Idle, что, в свою очередь, вызывает удаление интерфейсом сигнала Valid.

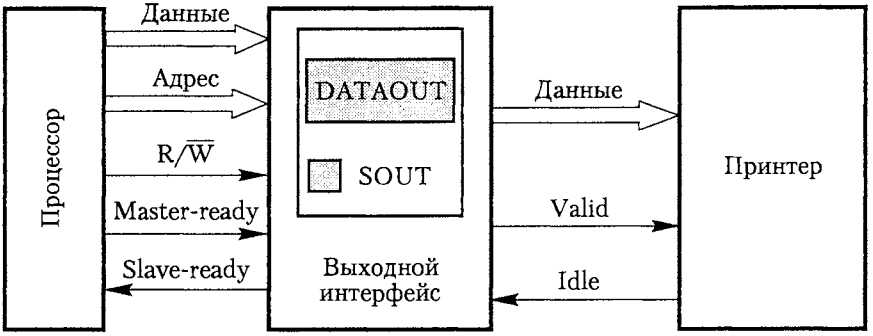


Рис. 4.31. Схема соединения процессора и принтера

Интерфейс имеет регистр данных DATAOUT и флаг состояния SOUT. Флаг устанавливается в 1, если принтер готов принять очередной символ, и очищается, когда новый символ загружается процессором в регистр DATAOUT. Реализация этого интерфейса показана на рис. 4.32. Принцип действия данной схемы аналогичен принципу действия схемы, приведенной на рис. 4.29. Единственным важным различием между ними является схема интерфейса, разработку которой мы оставляем за читателем в качестве полезного упражнения.

Рассмотренные нами интерфейсы ввода и вывода можно объединить в один (рис. 4.33). Этот общий интерфейс задается 30 старшими разрядами адреса. Для выбора одного из трех адресуемых регистров интерфейса (двух регистров данных и регистра состояния) предназначены адресные линии A1 и A0. В регистре состояния содержатся флаги SIN и SOUT, которым соответствуют разряды 0 и 1. Входы A1 и A0 мы обозначаем как RS0 и RS1 (от Register Select — выбор регистра), показывая таким образом, для чего они используются.

Предлагаемая схема имеет раздельные входные и выходные линии данных для соединения с устройством ввода-вывода. Если линии данных, ведущие к устройству ввода-вывода, являются двунаправленными, параллельный порт получается более гибким. На рис. 4.34 представлена универсальная схема параллельного интерфейса, которую можно конфигурировать множеством способов. Линии от P7 до P0 могут использоваться как для ввода, так и для вывода данных. С целью обеспечения большей гибкости схема даже позволяет использовать часть линий только для ввода, а остальные линии — только для вывода. Выбор линий осуществляется программным путем. Регистр DATAOUT соединяется с указанными линиями через повторители с тремя состояниями, которые управляются регистром, определяющим направление передачи данных DDR (Data Direction Register).

Процессор может записать в регистр DDR 8-разрядную маску. Если некоторый разряд этого регистра содержит значение 1, линия данных функционирует как выходная; в противном случае — как входная.

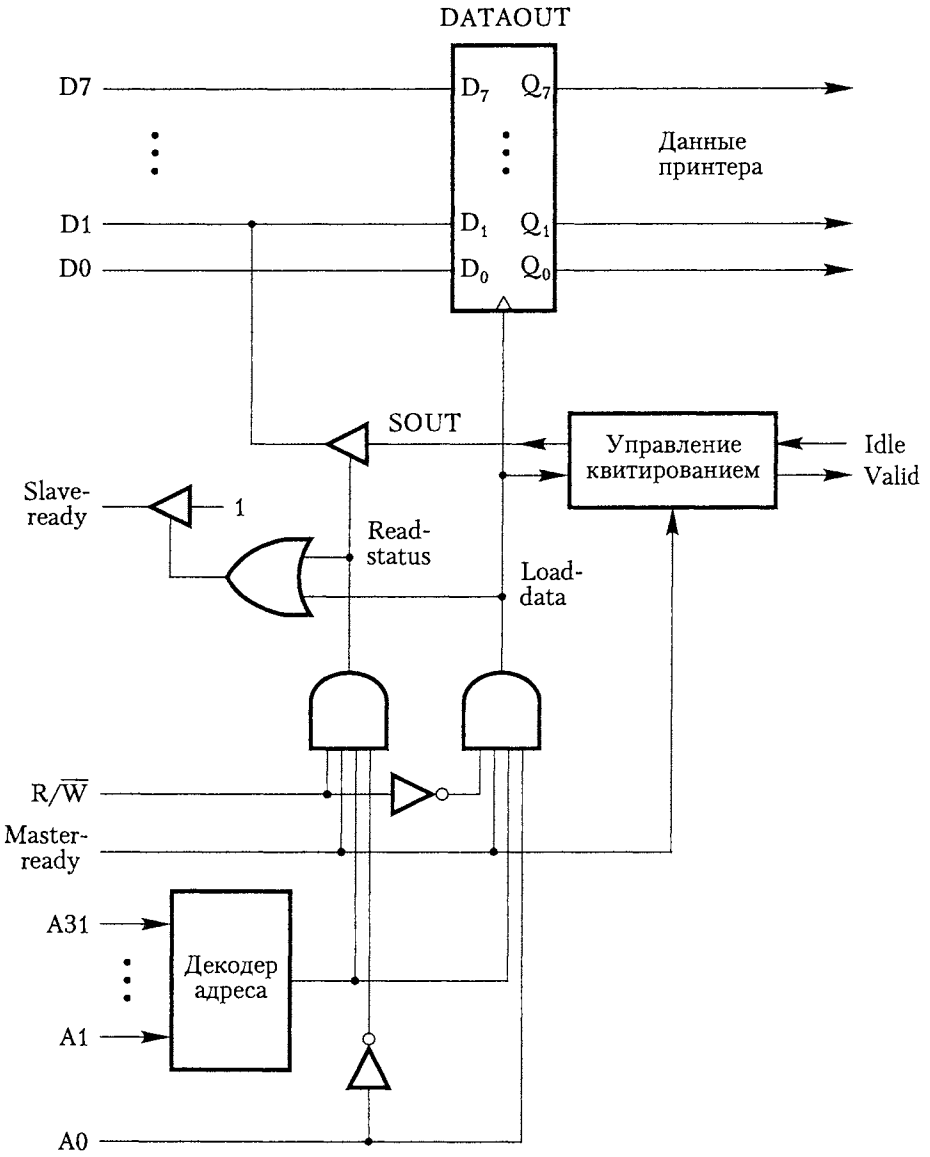


Рис. 4.32. Схема интерфейса выходного устройства

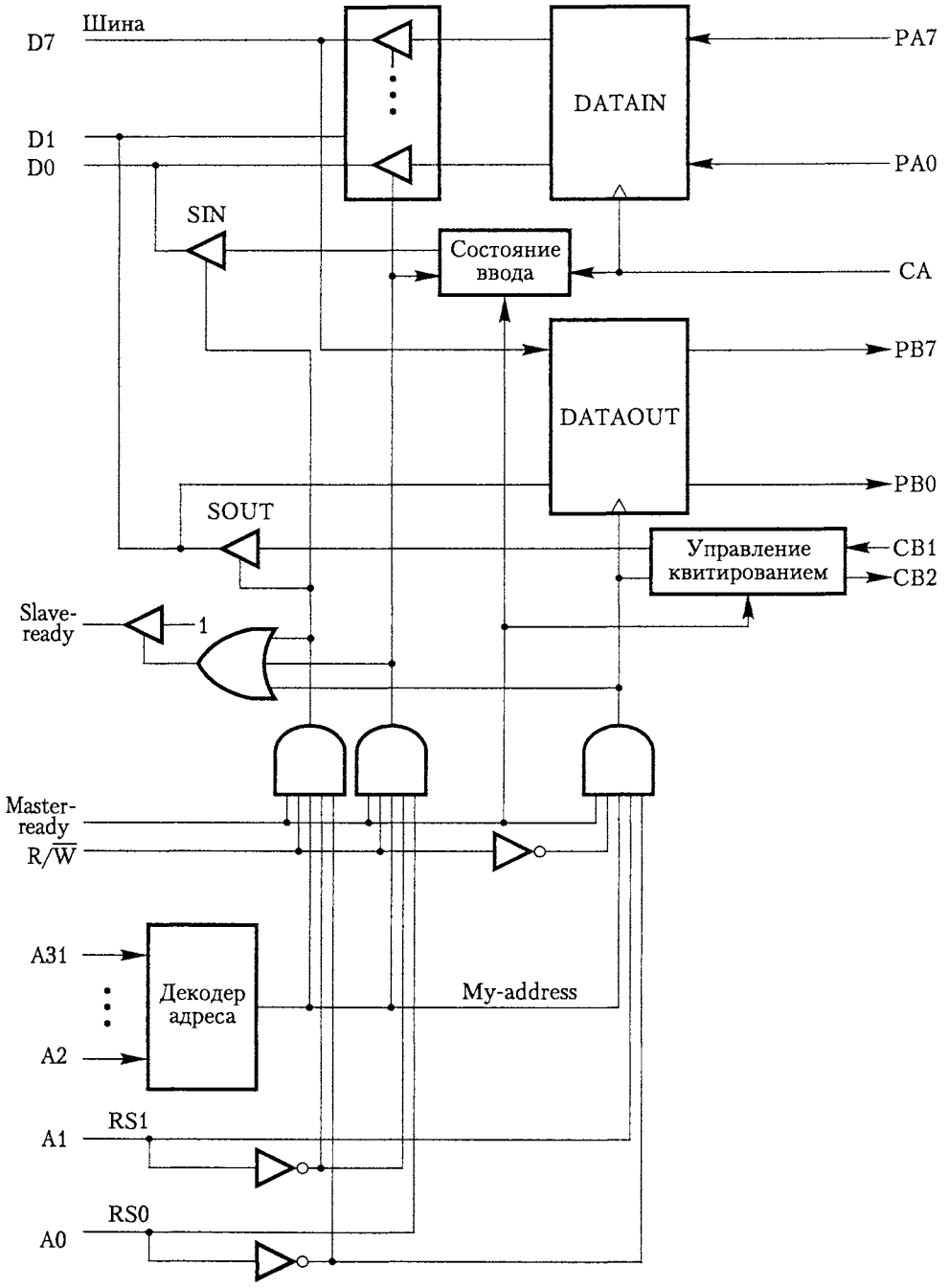


Рис. 4.33. Обобщенная схема интерфейса устройства ввода-вывода

Две линии, C1 и C2, управляют взаимодействием интерфейсной схемы и подключенного к ней устройства ввода-вывода. Эти линии также являются программируемыми. Двунаправленная линия C2 поддерживает несколько сигнальных режимов, включая и режим квитирования. Представленная на рис. 4.34 схема недостаточно подробна, тем не менее ее нетрудно сопоставить со схемой, которую вы видите на рис. 4.33.

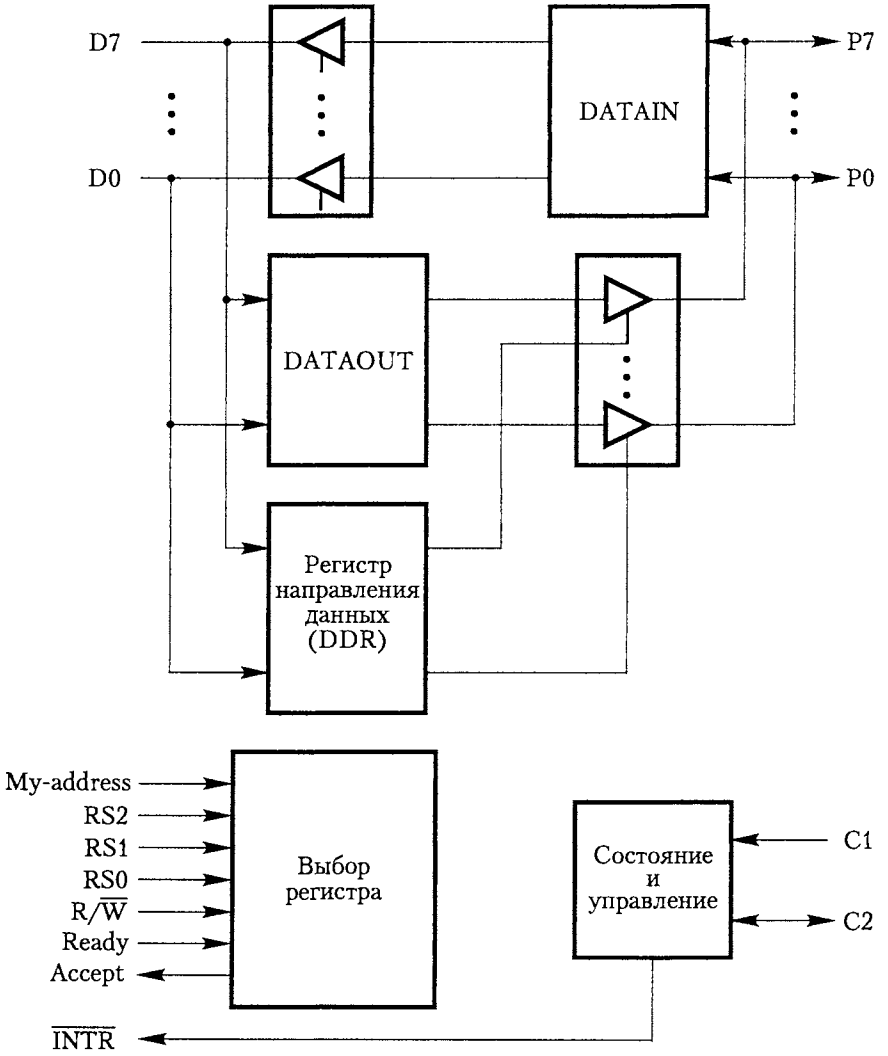


Рис. 4.34. Универсальный 8-разрядный параллельный интерфейс

Линии Ready и Accept используются для квитирования со стороны процессора, и их можно соединить с линиями Master-ready и Slave-ready. Входной сигнал My-address должен быть соединен с выходом декодера, который распознает адрес, назначенный данному интерфейсу. Существует три линии выбора регистра, позволяющие адресовать до восьми регистров интерфейса: регистры входных и выходных данных, регистр направления данных, управляющий регистр и регистр состояния для различных режимов функционирования схемы. Кроме того, имеется выход запроса прерывания $\overline{\text{INTR}}$, который должен быть соединен с линией запроса прерывания шины компьютера.

Схемы параллельного интерфейса, подобные показанной выше, встречаются довольно часто. Пример их применения во встроенной системе описан в главе 9. Вместо одного порта для подключения устройства ввода-вывода такая схема может включать два и более портов.

Теперь давайте посмотрим, как можно изменить интерфейсные схемы, представленные на рис. 4.28–4.34, для работы с синхронным шинным протоколом (рис. 4.25) Модифицированная интерфейсная схема с рис. 4.32 приведена на рис. 4.35. Мы добавили в нее тактовый логический блок, генерирующий сигналы Load-data и Read-status. В нижней части рисунка приведена диаграмма состояний этого блока. Сначала схема находится в состоянии Idle. Когда на выходе декодера адреса, обозначенном как My-address, появляется сигнал, означающий, что данный интерфейс адресован другим устройством, состояние схемы меняется на Respond. В результате она выдает сигнал Go, в ответ на который генерируется сигнал Load-data или Read-status, что зависит от значения адресного разряда A0 и состояния линии $R/\overline{W}$.

Временная диаграмма операции вывода данных приведена на рис. 4.36. На такте 1 процессор одновременно помещает на шину данные и адрес. В начале такта 2 тактовый логический блок устанавливает сигнал Go в 1 и на переднем фронте этого сигнала выходные данные загружаются в регистр DATAOUT. Операция ввода, считывающая значение регистра состояния, выполняется аналогично. Тактовый логический блок переходит из состояния Idle в состояние Respond, поскольку запрошенные данные помещены в регистр и могут быть немедленно переданы устройству. В результате пересылка данных выполняется на один такт быстрее, чем показано на рис. 4.25. В ситуации, когда данные становятся доступными через какое-то время, схема должна сначала перейти в состояние ожидания, и только дождавшись готовности данных — в состояние Respond.

Завершая обсуждение примеров интерфейсных схем, следует отметить, что мы использовали упрощенные представления некоторых сигналов. На практике для сигнала Slave-ready, скорее всего, будет использоваться выход с открытым стоком, обозначаемый как $\overline{\text{Slave-ready}}$, — по той же причине, что и сигнал $\overline{\text{INTR}}$. Эта линия должна включать нагрузочный резистор, на тот случай, если она не будет установлена каким-либо устройством в 1 (для чего ее напряжение необходимо понизить), то чтобы всегда находилась в состоянии логического 0 (которое требует высокого напряжения).

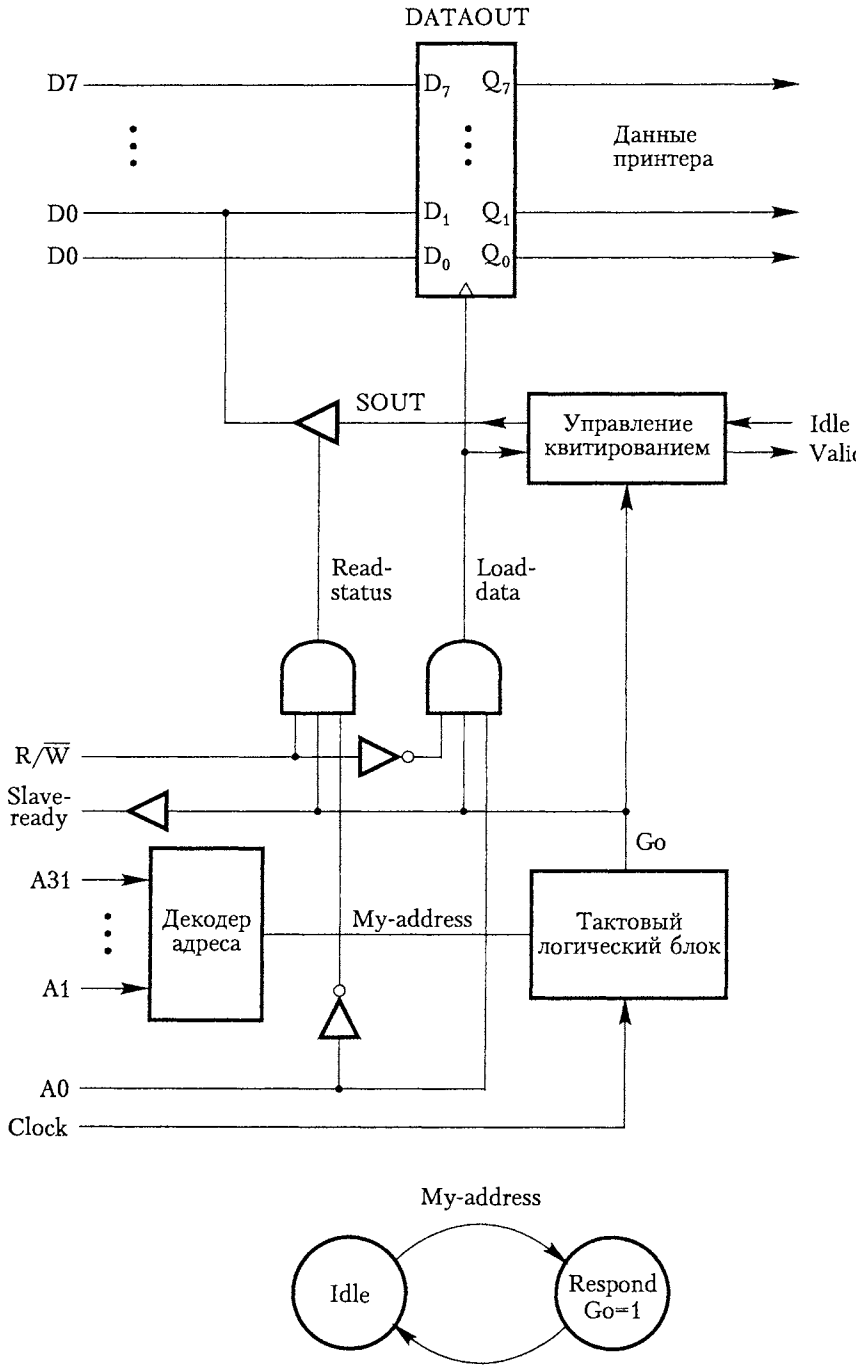


Рис. 4.35. Интерфейс параллельного порта для шины, показанной на рис. 4.25, и диаграмма состояния тактового логического блока

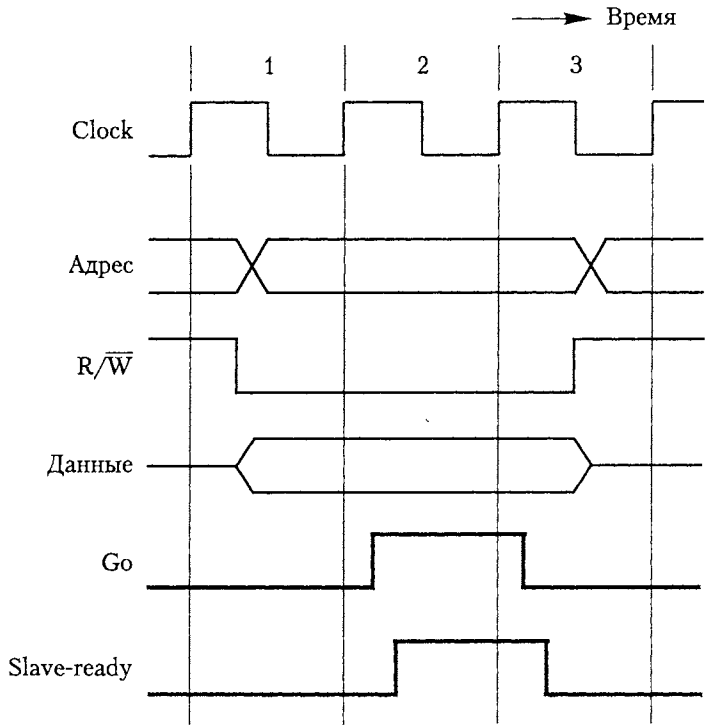


Рис. 4.36. Временная диаграмма интерфейсной схемы, приведенной на рис. 4.35

4.6.2. Последовательный порт

Последовательный порт используется для соединения процессора с устройствами ввода-вывода, которые передают данные по одному биту за раз. Важной особенностью интерфейсной схемы последовательного порта является то, что она способна передавать данные в последовательном режиме со стороны устройства и в параллельном режиме со стороны шины. Взаимопреобразование последовательных и параллельных форматов данных выполняется при помощи сдвиговых регистров, обладающих функцией параллельного доступа. На рис. 4.37 приведена блок-схема типичного последовательного интерфейса, включающая хорошо знакомые вам регистры DATAIN и DATAOUT. Входной сдвиговый регистр принимает от устройства ввода-вывода последовательные биты. После получения всех 8 бит данных содержимое этого регистра в параллельном режиме загружается в регистр DATAIN. Аналогичным образом выходные данные из регистра DATAOUT загружаются в выходной сдвиговый регистр, откуда биты по очереди отправляются устройству ввода-вывода.

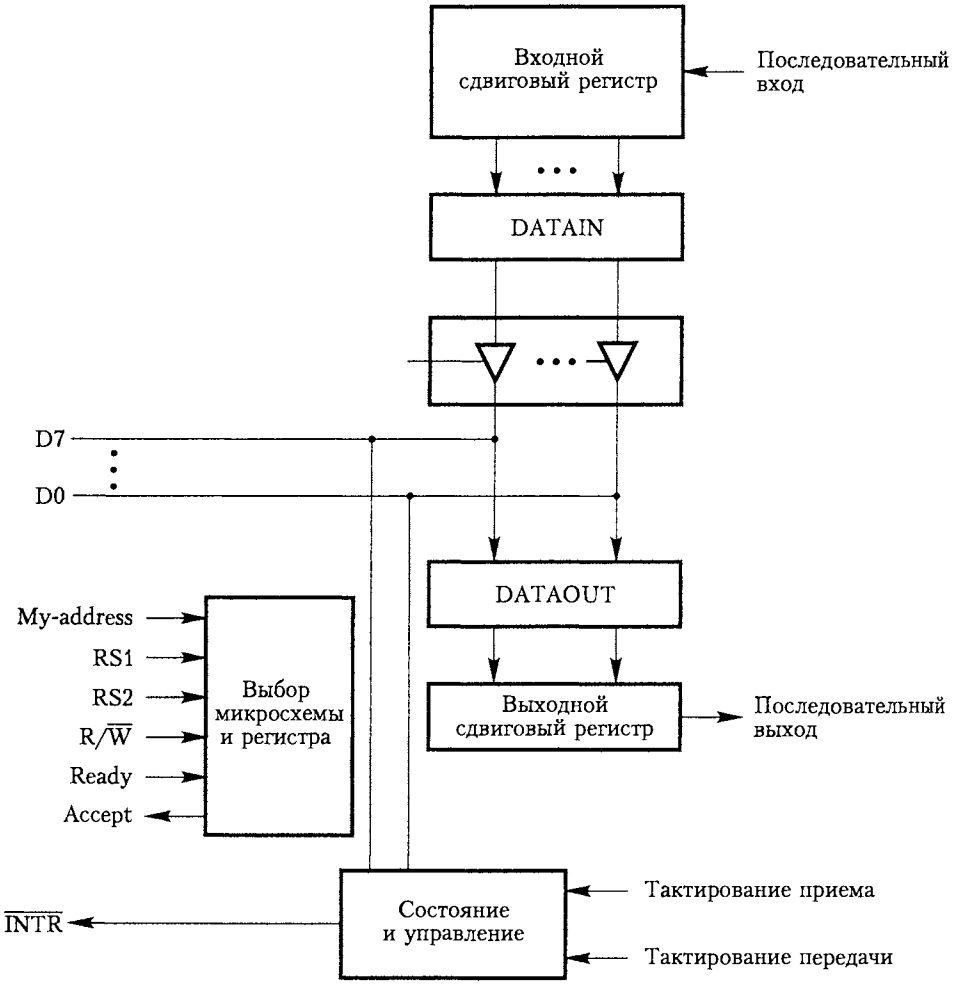


Рис. 4.37. Последовательный интерфейс

Та часть интерфейсной схемы, которая взаимодействует с шиной, мало чем отличается от описанного выше параллельного интерфейса. Флаги SIN и SOUT выполняют сходные функции. Флаг SIN устанавливается в 1, когда новые данные загружаются в регистр DATAIN, а когда процессор считывает содержимое этого регистра, значение флага SIN сбрасывается в 0. Как только данные пересылаются из входного сдвигового регистра в DATAIN, сдвиговый регистр может начать прием от устройства ввода-вывода следующего 8-битового символа. Флаг SOUT указывает, доступен ли выходной буфер для записи. Когда процессор записывает в DATAOUT новые данные, этот флаг очищается, а когда данные из DATAOUT перемещаются в сдвиговый регистр, SOUT устанавливается в 1.

Двойная буферизация данных на входе и выходе играет очень важную роль. Можно было бы упростить интерфейс, превратив буферы DATAIN и DATAOUT

в сдвиговые регистры и вовсе удалив исходные сдвиговые регистры, показанные на рис. 4.37. Однако тем самым мы наложили бы на функционирование устройства ввода-вывода неприемлемые ограничения. Так, после отправки одного символа устройство ввода не могло бы начать отправку следующего символа до тех пор, пока процессор не прочитал бы содержимое регистра DATAIN. А для того чтобы процессор мог читать входные данные, между двумя символами была бы необходима пауза. Если же буферы дублируются, пересылка второго символа может начитаться сразу же после загрузки первого символа из сдвигового регистра в регистр DATAIN. Если процессор считывает содержимое регистра DATAIN до завершения последовательной пересылки второго символа, интерфейс может принимать непрерывный поток входных данных. Аналогичный процесс происходит и на выходе интерфейса.

Поскольку для последовательной передачи данных требуется меньшее количество проводов, она удобна для применения в устройствах, которые физически находятся на значительном расстоянии от компьютера. Скорость пересылки данных, часто измеряемая в битах, зависит от конкретного устройства. Для того чтобы последовательный интерфейс мог работать с различными устройствами, он должен поддерживать разную тактовую частоту. Большой гибкостью, в частности, обладает схема, приведенная на рис. 4.37. Объясняется это тем, что она позволяет использовать для операций ввода и вывода раздельные тактовые сигналы.

Поскольку последовательный интерфейс используется для подключения к компьютерам огромного количества устройств ввода-вывода, для него разработано несколько популярных стандартов. Схема такого типа, как на рис. 4.37, известна под названием UART (Universal Asynchronous Receiver Transmitter — универсальный асинхронный приемопередатчик). Она предназначена для использования асинхронного стартстопного режима, о котором рассказывается в главе 10. Для коммуникационных соединений разработан другой популярный стандарт, называемый RS-232-C. О нем мы поговорим также в главе 10.

4.7. Стандартные интерфейсы ввода-вывода

В предыдущих разделах упоминалось о существовании нескольких альтернативных конструкций шины компьютера. Поэтому устройства ввода-вывода, подходящие для интерфейсной схемы одного компьютера, могут не подойти другому. Если бы разработчики компьютеров пошли по пути создания специального интерфейса для каждой новой пары, состоящей из внешнего устройства и компьютера, получилось бы невероятное количество разных интерфейсов. Вот почему разрабатываются стандартные интерфейсы, сигналы и протоколы, позволяющие подключать одни и те же внешние устройства к самым разным компьютерам.

Здесь очень важно понимать, что собой представляет компьютерная система и как соединяются между собой ее компоненты. Например, необходимо знать, что типичный персональный компьютер состоит из большой печатной платы, называемой материнской. На ней располагается микросхема процессора, основная память, несколько интерфейсов ввода-вывода, а также имеется несколько разъемов для подключения дополнительных интерфейсов.

Шина процессора — это шина, управляемая теми же сигналами, что и микросхема процессора. К ней могут быть подключены устройства, которым требуется очень высокая скорость взаимодействия с процессором, и в частности основная память. Из-за некоторых ограничений электрической природы с процессором может быть соединено лишь несколько устройств. На материнской плате обычно имеется еще одна шина, способная поддерживать большее количество устройств. Эти две шины соединены между собой с помощью специальной схемы, называемой *мостом* и предназначенной для преобразования сигналов в соответствии с протоколами, регулирующими применение этих двух шин. Устройства, подключенные к шине расширения, представляются процессору непосредственно соединенными с его собственной шиной. Правда, мост вызывает небольшую задержку при пересылке данных между процессором и этими устройствами.

Универсальный стандарт для шины процессора определить невозможно, поскольку ее структура очень тесно связана с архитектурой процессора. Структура шины зависит от электрических характеристик процессора, в том числе от его тактовой частоты. Однако на шину расширения эти ограничения не распространяются, поэтому для нее можно использовать стандартную схему сигналов. Для шин расширения разработан целый ряд стандартов. Некоторые из них появлялись «естественным» путем, когда конкретная архитектура завоевывала популярность на рынке. Например, IBM разработала для своего персонального компьютера шину ISA (Industry Standard Architecture), которая поначалу называлась PC AT. Компьютер стал настолько популярным, что производители устройств ввода-вывода стали снабжать свои устройства ISA-совместимыми интерфейсами, и ISA стал стандартом де-факто.

Некоторые стандарты разрабатывались объединенными усилиями крупных компаний, которые, хотя и конкурировали на рынке вычислительной техники, были заинтересованы в создании совместимых устройств. В некоторых случаях эти стандарты, одобренные такими организациями, как IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и электронике), ANSI (American National Standards Institute — Национальный институт стандартизации США), и даже международными организациями, в частности ISO (International Organization for Standardization — Международная организация по стандартизации), получали официальный статус.

В этом разделе вы познакомитесь с тремя широко применяемыми стандартами шин: PCI (Peripheral Component Interconnect), SCSI (Small Computer Systems Interface) и USB (Universal Serial Bus). На рис. 4.38 показано, как шины этих трех типов используются в типичной компьютерной системе. Стандарт PCI определяет шину расширения на материнской плате. Шины стандарта SCSI и USB предназначены для подключения дополнительных устройств как внутри, так и вне корпуса компьютера. SCSI представляет собой высокоскоростную параллельную шину, предназначенную для подключения таких устройств, как диски и дисплеи. Шина USB поддерживает последовательную передачу данных. Она используется для подключения самого разнообразного оборудования, от клавиатур до игровых устройств, а также для Интернет-соединений. На рисунке показана интерфейсная схема, позволяющая подключать к компьютеру устройства, совместимые со

старым стандартом ISA, в том числе столь популярные диски IDE (Integrated Drive Electronics). Данная шина может быть использована и для подключения компьютера к сети Ethernet (Ethernet — это широко распространенная архитектура локальных сетей, обеспечивающая высокоскоростное соединение компьютеров в здании или, предположим, университетском городке).

В одном компьютере может использоваться несколько разных шин. Так, типичный компьютер Pentium содержит шины PCI и ISA, что позволяет пользователю выбирать из достаточно широкого диапазона устройств.

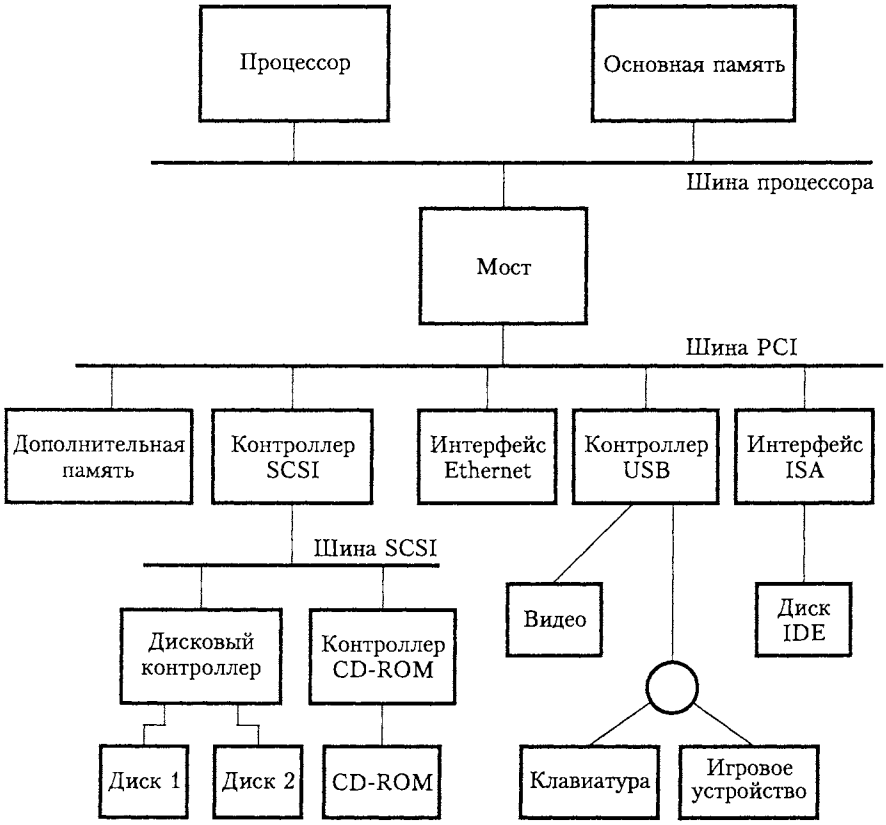


Рис. 4.38. Пример компьютерной системы, в которой используется несколько стандартов интерфейса

4.7.1. Шина PCI

Шина PCI — это разновидность системной шины, появившейся в ответ на потребность в стандартизации используемых устройств. Она поддерживает функции, типичные для шины процессора, но в стандартизированном формате, независимо от типа процессора. Подключенные к этой шине устройства представляются процессору непосредственно соединенными с его собственной шиной. Им назначаются

адреса из адресного пространства памяти процессора. (Полную спецификацию на шину, PCI Local Bus Specifications, вы найдете по адресу www.pcisig.com/developers.)

Шина PCI унаследовала многие принципы шинных стандартов, применявшихся преимущественно в компьютерах IBM PC. В ранних PC использовалась 8-разрядная шина XT, сигналы которой были очень близки к сигналам процессоров Intel 80*x*86. Более поздняя, 16-разрядная шина, которая в свое время предназначалась для компьютеров PC AT, известна как шина ISA. В 1980-е годы были разработаны и другие шины со сходными возможностями, и наиболее известными среди них считались Microchannel и NuBus, использовавшиеся соответственно в компьютерах IBM PC и Macintosh.

Создавалась шина PCI как недорогое устройство, по-настоящему независимое от процессора. Ее конструкция была обусловлена назревшей потребностью в поддержке высокоскоростных дисковых и графических устройств, а также специфическими нуждами мультипроцессорных систем. Благодаря этому PCI до сих пор, десятилетие спустя после своего появления в 1992 году, популярна как промышленный стандарт.

Важной особенностью шины PCI было то, что она стала пионером нового механизма подключения устройств ввода-вывода, получившего название plug-and-play. Для подключения к системе нового устройства пользователю теперь достаточно вставить интерфейсную плату в разъем на шине. Все остальное сделает за него программное обеспечение. Мы вернемся к технологии plug-and-play после обсуждения основных принципов работы шины PCI.

Пересылка данных

В современных компьютерах при выполнении операции пересылки данных чаще всего перемещается не одно слово, а целый блок информации. Поэтому все современные процессоры содержат кэш-память (см. рис. 1.6). Данные пересылаются между кэш-памятью и основной памятью в виде пакетов, по нескольку слов в каждом (об этом подробнее рассказывается в главе 5). Участвующие в такой пересылке слова сохраняются по последовательным адресам памяти. Когда процессор, а точнее кэш-контроллер, задает адрес и запрашивает операцию чтения из основной памяти, память отвечает ему отправкой последовательности слов данных, начинающихся с этого адреса. Аналогичным образом в ходе операции записи процессор задает адрес и последовательность слов данных, которые должны быть поочередно записаны в память, начиная с этого адреса. Шина PCI предназначена для поддержки именно такого режима работы. Операция чтения или записи одного слова интерпретируется ею как чтение или запись пакета длиной в одно слово.

Шина поддерживает три независимых адресных пространства: памяти, ввода-вывода и конфигурации. Назначение первых двух понятно. Адресное пространство ввода-вывода используется такими процессорами, как Pentium, имеющими отдельное адресное пространство ввода-вывода. Однако, как уже было подчеркнуто в главе 3, конструктор системы может использовать ввод-вывод с отображением в память даже в том случае, если процессор поддерживает отдельное адресное пространство ввода-вывода. Фактически стандартом PCI для совместимости с более широким спектром устройств рекомендуется применять

именно этот подход. Конфигурационное адресное пространство предназначено для поддержки технологии plug-and-play. Сопровождающая адрес 4-разрядная команда указывает, какое из трех адресных пространств должно использоваться в этой операции пересылки данных.

На рис. 4.38 показана схема, на которой основная память компьютера непосредственно соединена с шиной процессора. Альтернативный подход, часто используемый при наличии шины PCI, проиллюстрирован на рис. 4.39. Мост PCI создает для основной памяти отдельное физическое подключение. По причинам электротехнического характера шина может быть разделена на сегменты, соединенные между собой посредством мостов. Однако независимо от того, к какому сегменту шины подключено конкретное устройство, оно может отображаться в адресное пространство процессора.

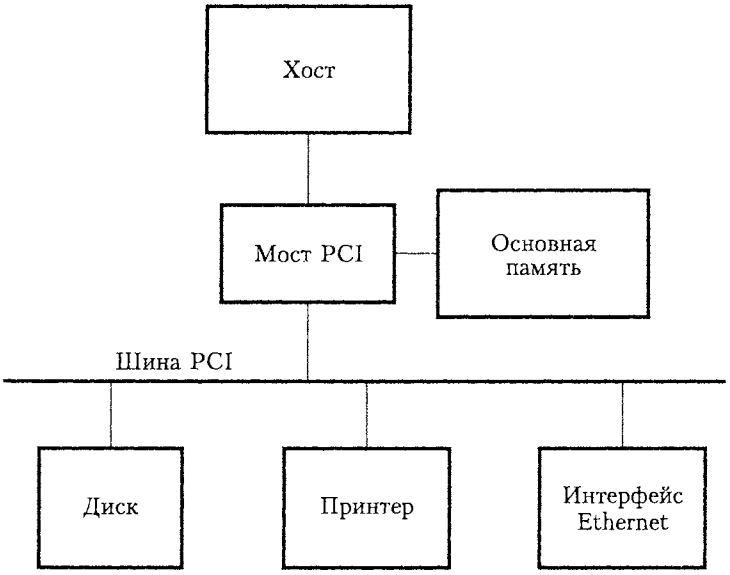


Рис. 4.39. Пример использования шины PCI в компьютерной системе

Набор сигналов шины PCI похож на набор сигналов, представленный на рис. 4.25. Создавая этот рисунок, мы предполагали, что хозяин шины сохраняет на ней адресную информацию до тех пор, пока пересылка данных не завершится. Однако так бывает не всегда. Адрес нужен только до тех пор, пока не выбрано подчиненное устройство, которое теперь может хранить адрес в своем внутреннем буфере. Таким образом, адрес должен находиться на шине в течение одного такта, после чего адресные линии могут быть освобождены для пересылки данных в последующих тактах. Благодаря этому снижается стоимость операции пересылки, напрямую зависящая от количества проводов шины. Этот подход и используется в шине PCI.

Хозяином шины в каждый конкретный момент времени может быть только одно устройство. Это устройство имеет право инициировать операции пересылки

данных с помощью команд чтения и записи. Согласно терминологии PCI, хозяин шины называется *инициатором*. Им может быть либо процессор, либо контроллер ПДП. Адресуемое устройство, отвечающее на команды чтения и записи, называется *целевым*.

Чтобы понять, как работает шина, нужно рассмотреть типичную операцию пересылки. Основные сигналы шины, используемые для пересылки данных, перечислены в табл. 4.3. Для сигналов, имена которых начинаются с символа «#», активным является низковольтное состояние. Главное отличие протокола PCI от схемы, показанной на рис. 4.25, заключается в том, что в дополнение к сигналу готовности целевого устройства TRDY# в нем используется сигнал готовности инициатора IRDY#. Последний необходим для поддержки пакетной пересылки данных.

Таблица 4.3. Сигналы пересылки данных по шине PCI

Имя	Описание
CLK	Тактовый сигнал с частотой 33 или 66 МГц
FRAME#	Активизируется инициатором с целью определения длительности транзакции
AD	Представляет 32 линии для пересылки адресов и данных (количество линий при необходимости может быть увеличено до 64)
C/BE#	Представляет 4 линии для команды считывания (Command) и массив, указывающий, какие байты подлежат считыванию (Byte enable)
IRDY#, TRDY#	Сигналы готовности инициатора и целевого устройства
DEVSEL#	Ответ устройства, указывающий, что оно распознало свой адрес и готово к операции пересылки данных
IDSEL#	Сигнал выбора инициализируемого устройства

Давайте проанализируем операцию, в ходе которой процессор считывает из памяти 32-разрядное слово. Ее инициатором является процессор, а целевым устройством — память. Полная операция пересылки данных по шине, включающая пересылку адреса и пакета данных, называется *транзакцией*. Пересылка отдельного слова в ходе транзакции называется *фазой*. Последовательность событий шины показана на рис. 4.40. Тактовый сигнал используется для координирования различных фаз транзакции. Все изменения сигналов инициируются передним фронтом тактового сигнала. Как и на рис. 4.25, из-за задержек изменения сигналов следуют через некоторое время после тактовых импульсов.

На такте 1 процессор помещает на шину сигнал FRAME#, сообщая тем самым о начале транзакции. Одновременно он помещает адрес на линии AD и команду на линии C/BE#. В данном случае команда указывает, что запрошена операция чтения и что в ней используется адресное пространство памяти.

Такт 2 предназначен для переключения линий шины AD. Процессор удаляет с них адрес и отключает свои выходные повторители. Выбранное целевое устройство, напротив, включает свои повторители на линиях AD и в течение такта 3 помещает на эти линии запрошенные данные. Затем оно активизирует сигнал DEVSEL# и поддерживает его в активном состоянии до конца транзакции.

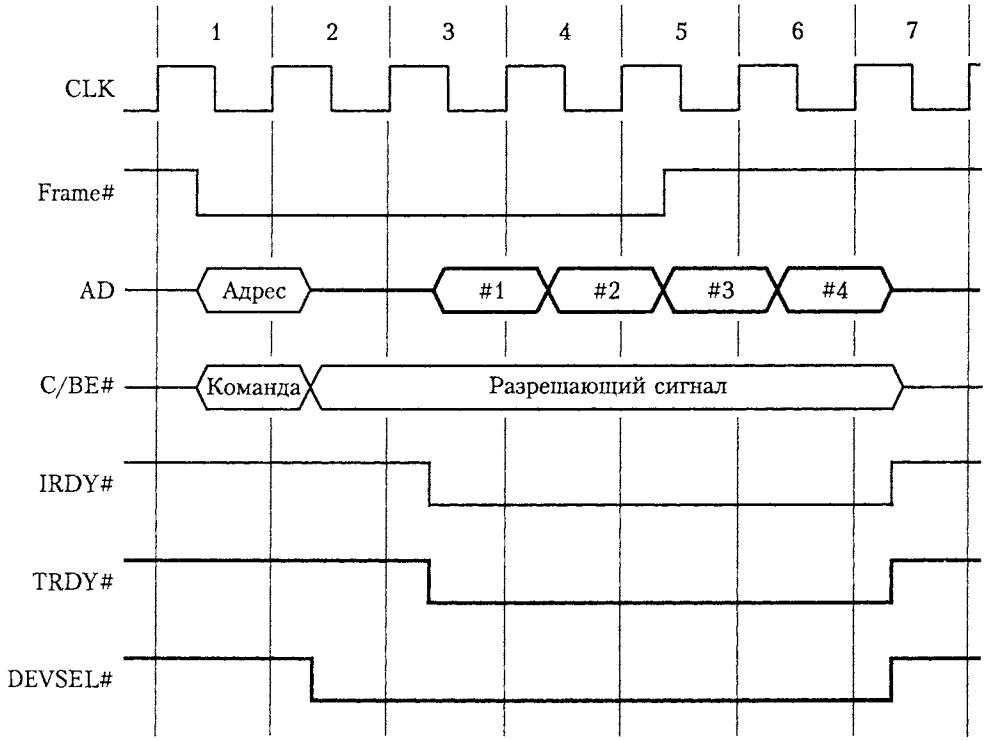


Рис. 4.40. Операция чтения на шине PCI

Линии C/BE#, использовавшиеся для пересылки команды на такте 1, в оставшейся части транзакции применяются для другой цели. Каждая из них связана с одним байтом данных на линиях AD. Инициатор активизирует одну или несколько линий C/BE#, для того чтобы указать, какие из линий AD должны использоваться для пересылки данных. Если целевое устройство способно одновременно пересылать 32 бита данных, все четыре линии C/BE# устанавливаются в 1.

В течение такта 3 инициатор активизирует сигнал IRDY#, с тем чтобы сообщить о своей готовности к получению данных. Если целевое устройство к этому моменту уже готово к отправке данных, оно устанавливает сигнал TRDY# и помещает на шину слово данных. В конце такта инициатор загружает данные в свой входной буфер. На тактах 4–6 целевое устройство отсылает инициатору еще три слова данных.

С помощью сигнала FRAME# инициатор указывает длину пакета данных. Во время передачи предпоследнего слова он снимает этот сигнал, отмечая окончание пакета. В нашем примере инициатор хочет прочитать четыре слова. Поэтому он снимает сигнал FRAME# на такте 5, во время которого получает третье слово данных. После отправки четвертого слова на такте 6 целевое устройство отсоединяет свои повторители от линий AD и в начале такта 7 снимает сигнал DEVSEL#.

На рис. 4.41 приведен более обобщенный пример транзакции ввода данных. Он показывает, как с помощью сигналов IRDY# и TRDY# инициатор и целевое

устройство делают паузы в середине транзакции. Целевое устройство отсылает третье слово на такте 5. Предположим, что в этот момент инициатор не готов к его получению. Тогда он снимает сигнал IRDY#, а целевое устройство сохраняет третье слово на линях AD до тех пор, пока сигнал IRDY# не появится снова. На такте 6 инициатор активизирует сигнал IRDY# и в конце этого такта загружает данные в свой входной буфер. Допустим, что теперь целевое устройство не готово к немедленной отправке четвертого слова данных. В начала такта 7 оно снимает сигнал TRDY#, а на такте 8 помещает четвертое слово на линии AD и активизирует сигнал TRDY#. Поскольку на третьем слове данных сигнал FRAME# снимается, после пересылки четвертого слова данных транзакция завершается.

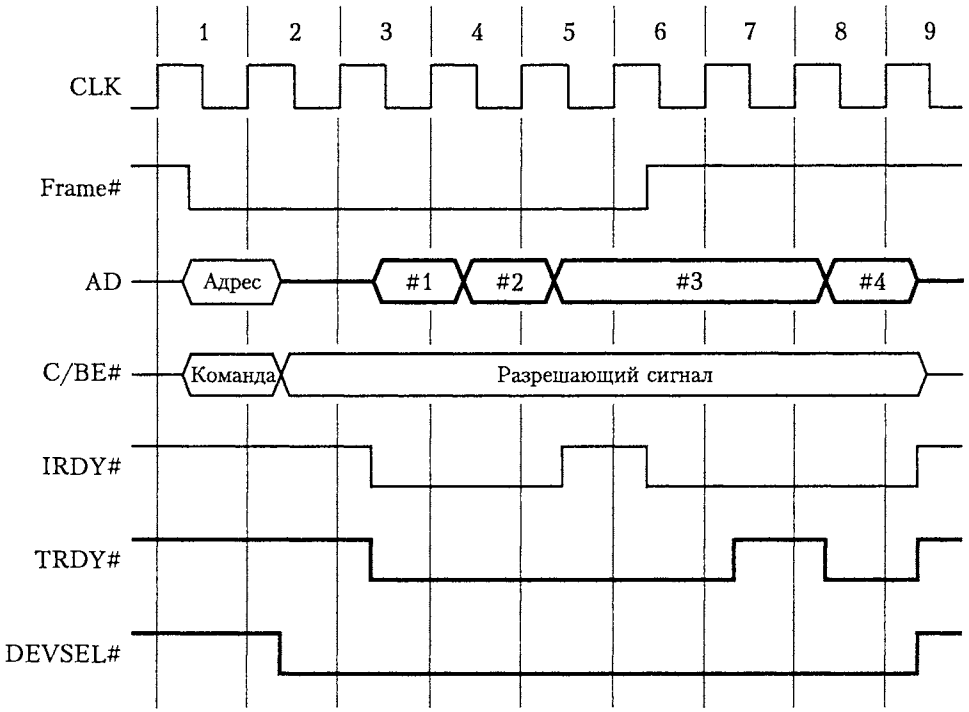


Рис. 4.41. Роль сигналов IRDY# и TRDY# в операции чтения

Конфигурирование устройства

Когда устройство ввода-вывода подключено к компьютеру, для настройки этого устройства и взаимодействующего с ним программного обеспечения выполняется несколько действий. Например, на типичной интерфейсной плате устройства для шины ISA имеется множество перемычек или переключателей, устанавливаемых пользователем для выбора определенных опций. Если устройство подключено к компьютеру, программному обеспечению нужно знать его адрес. Кроме того, ему может понадобиться информация о различных характеристиках устройства, таких как скорость работы канала связи, использование битов четности и т. д.

Стандарт PCI упрощает этот процесс путем включения в интерфейс каждого устройства ввода-вывода конфигурационной ROM-памяти небольшого объема, предназначенной для хранения информации об устройстве. При включении или перезапуске компьютера программное обеспечение, выполняющее инициализацию шины PCI, считывает содержимое этой памяти на основании полученных данных определяет тип каждого устройства (принтер, клавиатура, сетевая плата, дисковый контроллер и т. д.) и, если нужно, узнает опции и характеристики устройства.

В результате процесса инициализации устройствам назначаются адреса. Это означает, что в ходе операции конфигурирования шины доступ к устройствам по их адресам не возможен, поскольку таковые им пока еще не назначены. Поэтому для доступа к адресному пространству конфигурации применяется другой механизм. У каждого устройства имеется входной сигнал, называемый IDSEL# (Initialization Device Select — выбор инициализируемого устройства). В ходе операции конфигурирования выбор устройства выполняется с помощью этого сигнала, а не с помощью адреса, поданного на AD-входы устройства. На материнской плате в каждом разъеме, к которому подключается интерфейс устройства ввода-вывода, имеется контакт IDSEL#, соединенный с одной из 21 старших адресных линий, от AD11 до AD31. Выбор устройства для конфигурирования выполняется путем подачи на шину конфигурационной команды и адреса, в котором соответствующая линия AD установлена в 1, а оставшиеся 20 линий — в 0. Младшие адресные линии, от AD10 до AD00, используются для задания типа операции и для доступа к содержимому конфигурационной ROM устройства. Такая организация шины PCI позволяет подключить к ней не более 21 устройства ввода-вывода.

Конфигурационное программное обеспечение сканирует 21 область конфигурационного адресного пространства, чтобы узнать, какие устройства подключены к компьютеру. Каждое устройство может запросить адрес в пространстве ввода-вывода или в пространстве основной памяти. Назначенный устройству адрес записывается в один из его регистров. Кроме того, конфигурационное программное обеспечение устанавливает такие параметры, как приоритет прерываний устройства. Шина PCI включает четыре линии запроса прерывания. Записывая данные в регистр конфигурации устройства, программное обеспечение информирует устройство о том, какие из этих линий оно может использовать для запроса прерывания. Если устройству необходима инициализация, предназначенный для этой цели код хранится в специальной ROM его интерфейса. (Это не та ROM, которая используется в процессе конфигурирования.) Программное обеспечение PCI считывает данный код и выполняет его для инициализации устройства.

Описанная процедура освобождает пользователя от участия в конфигурационном процессе. Теперь ему необходимо лишь вставить интерфейсную плату в разъем и включить питание. Все остальное делает программное обеспечение, по окончании работы которого устройство готово к использованию.

Шина PCI завоевала огромную популярность в мире персональных компьютеров PC. Используется она и в компьютерах других типов, в том числе в Sun, поскольку для нее имеется множество устройств ввода-вывода. В некоторых процессорах, и в частности в Compaq Alpha, даже имеется встроенный мост PCI–процессор, еще более упрощающий создание компьютерных систем.

Электрические характеристики

Шина PCI предназначена для работы под напряжением питания 5 или 3,3 В. Материнская плата поддерживает любую систему сигналов. Соединительные разъемы и платы расширения изготавливаются таким образом, чтобы их можно было подключать только к совместимым материнским платам.

4.7.2. Шина SCSI

Как уже было отмечено, SCSI — это аббревиатура от Small Computer Systems Interface, что в переводе с английского означает интерфейс малых компьютерных систем. Так называется стандарт, определенный Национальным институтом стандартизации США (American National Standards Institute, ANSI) под номером X3.131. Согласно основной спецификации этого стандарта, такие устройства, как диски, должны соединяться с компьютером при помощи 50-проводного кабеля длиной до 25 м, по которому данные могут передаваться со скоростью до 5 Мбайт/с.

Стандарт SCSI претерпел много изменений, и все это время определяемая им скорость передачи данных постоянно увеличивалась, почти удваиваясь каждые два года. К настоящему времени определены стандарты SCSI-2 и SCSI-3, каждый из которых имеет несколько опций. Шина SCSI может иметь восемь линий данных — в этом случае она называется узкой (narrow SCSI) и передает данные по одному байту за раз. Широкая шина SCSI (wide SCSI) состоит из 16 линий данных и передает информацию по 16 бит за раз. Существует несколько вариантов электрических сигнальных схем. Передача данных по шине SCSI может выполняться в асимметричном режиме (Single-Ended SCSI, SE), когда для каждого сигнала используется свой проводник с общим замыканием через «землю» для всех сигналов. В качестве альтернативы для каждого сигнала может предназначаться отдельный обратный провод. В этом случае возможно использование двух уровней напряжения. В ранних версиях шины, получивших название High Voltage Differential (HVD), применялось напряжение 5 В (уровни TTL). Более поздняя версия с напряжением 3,3 В называется Low Voltage Differential (LVD).

Для различных версий SCSI используются разные разъемы: они могут быть 50-, 68- и 80-контактными. Максимальная скорость передачи данных современных устройств варьируется от 5 до 60 Мбайт/с. Последняя версия стандарта поддерживает скорость передачи до 320 Мбайт/с, а на подходе версия с поддержкой 640 Мбайт/с. Максимальная скорость передачи по конкретной шине зависит от длины кабеля и количества подключенных к нему устройств. Но если говорить в общем, она тем выше, чем короче кабель и чем меньше устройств к нему подключено. Для достижения максимальной скорости обычно используют кабель длиной не более 1,6 м для сигнальной схемы SE и не более 12 м для сигнальной схемы LVD. Однако производители для подключения более удаленных устройств часто предоставляют специальные расширители шины. Максимальная «вместимость» шины составляет 8 устройств для Narrow SCSI и 16 устройств для Wide SCSI.

В отличие от устройств, подключаемых к шине процессора, устройства, подключаемые к шине SCSI, не являются частью его адресного пространства. Шина SCSI соединяется с шиной процессора через SCSI-контроллер, как показано на

рис. 4.38. Для пересылки пакетов данных от главной памяти к устройству и в обратном направлении этот контроллер применяет технологию прямого доступа к памяти. Пакет может содержать блок данных, команды, направляемые процессором устройству, или информацию о состоянии устройства.

Ниже функционирование шины SCSI будет рассмотрено на примере ее использования с дисковым накопителем. Принцип взаимодействия с дисками очень отличается от принципа взаимодействия с основной памятью. Как рассказывается в главе 5, данные хранятся на диске блоками, называемыми *секторами*, каждый из которых может содержать несколько сот байтов. Данные не обязательно записываются в последовательно расположенные секторы. Дело в том, что в одних секторах могут уже храниться ранее записанные данные; другие могут быть дефектными, а следовательно, должны быть пропущены. Поэтому для обслуживания запроса чтения или записи может потребоваться доступ к нескольким, не обязательно последовательным секторам диска. Из-за ограничений, связанных с механической природой диска, обращение к первому сектору, из которого считываются или в который записываются данные, выполняется с довольно значительной задержкой, порядка нескольких миллисекунд. После этого некоторый объем данных пересылается с очень высокой скоростью, но затем может произойти еще одна задержка и т. д. В ходе обслуживания одного запроса чтения или записи может произойти несколько таких задержек. Протокол SCSI ориентирован именно на такой режим работы.

Контроллер, подключенный к шине SCSI, может быть *инициатором* или *целевым устройством*. Инициатор обладает способностью выбирать конкретное целевое устройство и направлять ему команды, определяющие выполняемую операцию. Очевидно, что контроллер со стороны процессора (например, изображенный на рис. 4.38 SCSI-контроллер) должен функционировать как инициатор. Дисковый контроллер является целевым устройством. Он осуществляет команды, получаемые от инициатора. Инициатор устанавливает *логическое соединение* с выбранным целевым устройством. Соединение может временно разрываться и возобновляться по мере возникновения необходимости в пересылке команд и пакетов данных. Когда некоторое соединение временно разрывается, шина может быть использована для передачи информации другими устройствами. Эта способность чередовать запросы пересылки данных является одной из важнейших особенностей шины SCSI, определяющих ее высокую производительность.

Пересылка данных по шине SCSI всегда управляется целевым контроллером. Для того чтобы отправить ему команду, инициатор запрашивает управление шиной, выиграв арбитраж, выбирает контроллер, с которым хочет взаимодействовать, и передает ему управление шиной. После этого целевой контроллер начинает операцию передачи данных для получения команды от инициатора.

Давайте в качестве примера рассмотрим весь процесс выполнения операции чтения данных с диска. Мы будем говорить о том, что инициирующий контроллер выполняет определенные действия, но читатель должен понимать, что он делает это только после получения соответствующих команд от процессора. Предположим, что процессор хочет прочитать с диска блок данных и что сектора, в которых хранится этот блок, расположены непоследовательно. Процессор

направляет SCSI-контроллеру команду, в ответ на которую происходят следующие события.

1. Контроллер SCSI, как инициатор, запрашивает управление шиной.
2. Выиграв арбитраж, он выбирает целевой контроллер и передает ему управление шиной.
3. Целевой контроллер начинает операцию вывода, а инициатор направляет ему в ответ команду, определяющую операцию чтения.
4. Целевой контроллер, который вначале должен выполнить операцию поиска данных на диске, отсылает инициатору сообщение, указывающее, что он временно разрывает соединение. После этого он освобождает шину.
5. Целевой контроллер направляет диску команду переместить считывающую головку к первому сектору, содержащему запрошенные данные. Затем он считывает хранящиеся в этом секторе данные и сохраняет их в буфере данных. Когда контроллер готов начать пересылку данных инициатору, он запрашивает управление шиной. Выиграв арбитраж, он снова выбирает инициирующий контроллер, возобновляя тем самым временно разорванное соединение.
6. Целевой контроллер пересылает инициатору содержимое буфера данных и еще раз временно разрывает соединение. Данные пересылаются параллельно по 8 или 16 бит, в зависимости от ширины шины.
7. Целевой контроллер направляет диску команду выполнить еще одну операцию поиска. Затем он пересылает инициатору содержимое второго сектора диска. Когда пересылка завершается, логическое соединение между двумя контроллерами разрывается.
8. Получив данные, инициирующий контроллер сохраняет их в основной памяти с использованием ПДП.
9. Контроллер SCSI направляет процессору запрос прерывания, для того чтобы проинформировать его о завершении операции.

Обмен сообщениями по шине SCSI выполняется на более высоком уровне, нежели обмен сообщениями по шине процессора. В данном контексте слова «на более высоком уровне» означают, что сообщения относятся к операциям, которые в зависимости от устройства могут требовать выполнения целого ряда действий. Ни процессору, ни контроллеру SCSI не нужно знать всех подробностей выполнения конкретным устройством операций, связанных с пересылкой данных. В нашем примере процессору незачем участвовать в операциях поиска данных на диске.

Стандартом SCSI определяется множество управляющих сообщений, которыми могут обмениваться контроллеры для управления различными типами устройств ввода-вывода. Кроме того, им определяются сообщения для обработки различных ошибок и нестандартных ситуаций, которые могут возникать в ходе работы устройств и пересылки данных.

Сигналы шины

Мы описали функционирование шины SCSI с аппаратной точки зрения. Основные сигналы этой шины перечислены в табл. 4.4, правда, там указаны только сигналы для узкой шины (с 8 линиями данных). Обратите внимание, что имена всех

сигналов начинаются со знака «–». Это означает, что сигнал активен, то есть линия данных равна 1, когда она находится в низковольтном состоянии. Шина не имеет адресных линий. Для идентификации контроллеров в процессе выбора или повторного выбора и в ходе арбитража применяются линии данных. Узкая шина позволяет использовать до 8 контроллеров, нумеруемых цифрами от 0 до 7, каждому из которых соответствует линия данных с тем же номером. К широкой шине можно подключить до 16 контроллеров. Для того чтобы поместить на шину свой адрес или адрес другого контроллера, контроллер активизирует соответствующую линию данных. Таким образом, на шине может одновременно присутствовать несколько адресов, как в описанном ниже процессе арбитража. После установления соединения между двумя контроллерами адресация больше не требуется, поэтому линии данных используются для пересылки данных.

Таблица 4.4. Сигналы шины SCSI

Категория	Имя	Описание
Данные	От –DB(0) до –DB(7)	Линии данных: пересылка одного бита информации на фазе пересылки и идентификация устройства на фазах арбитража, выбора и повторного выбора
	–DB(P)	Бит четности для шины данных
Фаза	–BSY	Действует, когда шина не свободна
	–SEL	Действует во время выбора и повторного выбора
Тип информации	–C/D	Устанавливается во время пересылки управляющей информации (команды, состояния или сообщения)
	–MSG	Указывает, что пересылаемая информация представляет собой сообщение
Квитирование	–REQ	Устанавливается целевым устройством для запроса цикла пересылки данных
	–ACK	Активизируется инициатором после завершения им операции пересылки данных
Направление пересылки	–I/O	Активизируется для операции ввода (с точки зрения инициатора)
Прочее	–ATN	Устанавливается инициатором при необходимости послать сообщение целевому устройству
	–RST	Вызывает отключение от шины всех контроллеров и их переход в исходное состояние

В процессе обмена данными по шине SCSI можно выделить несколько фаз, главными из которых являются: арбитраж, выбор и пересылка информации, повторный ее выбор. Давайте рассмотрим каждую из указанных фаз подробнее.

Арбитраж

Шина свободна, когда сигнал –BSY находится в неактивном (высоковольтном) состоянии. В это время запрос на ее использование может прислать любой контроллер. А поскольку такой запрос могут одновременно сгенерировать несколько контроллеров, необходима арбитражная схема. Для того чтобы запросить управление шиной, контроллер активизирует сигнал –BSY и идентифицирует себя, активизируя соответствующую линию данных. Для шины SCSI используется простая схема распределенного арбитража, иллюстрируемая рис. 4.42, когда контроллеры 2 и 6 запрашивают управление шиной одновременно.

Каждому подключенному к шине контроллеру в соответствии с его номером назначается фиксированный приоритет. Наивысший приоритет имеет контроллер 7. Когда активизируется сигнал –BSY, каждый контроллер, запросивший управление шиной, анализирует линии данных и определяет, запросил ли управление шиной контроллер с более высоким приоритетом. Если нет, контроллер 7 считает, что он имеет наивысший приоритет среди претендентов, а значит, он и выиграл арбитраж. Остальные контроллеры отсоединяются от шины и ждут снятия сигнала –BSY.

В схеме представленной на рис. 4.42, предполагается, что контроллер 6 является инициатором, желающим установить соединение с контроллером 5. Выиграв арбитраж у устройства 2, контроллер 6 переходит к фазе выбора и идентифицирует целевой контроллер.

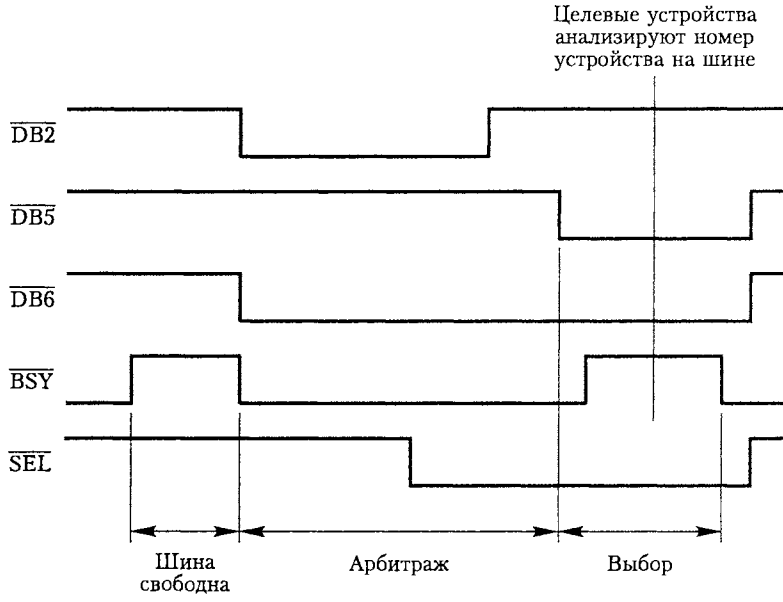


Рис. 4.42. Арбитраж и выбор на шине SCSI

Выбор

Выиграв арбитраж, контролер 6 сохраняет активными сигналы –BSY и –DB6 (свой адрес). Для того чтобы сообщить, что он хочет выбрать для соединения контроллер 5, он поочередно активизирует линии –SEL и –DB5. Как только активизируется сигнал –SEL, все остальные контроллеры, участвовавшие в арбитраже (подобно контроллеру 2 на рис. 4.42), снимают с линий данных свои адреса (если они этого еще не сделали). Поместив на шину адрес целевого контроллера, инициатор освобождает линию –BSY.

Выбранный целевой контроллер отвечает активизацией линии –BSY. Тем самым он сообщает инициатору, что запрошенное соединение установлено и что можно удалять с шины адрес целевого контроллера. На этом процесс выбора завершается, а целевой контроллер (контроллер 5) сохраняет линию –BSY активной. С этого момента шиной управляет целевой контроллер, что необходимо для фазы пересылки информации.

Передача данных

Информация, пересылаемая между двумя контроллерами, может состоять из команд, направляемых инициатором целевому контроллеру, информации о состоянии, направляемой целевым контроллером инициатору, и данных, пересылаемых между устройством ввода-вывода и инициатором в одном из двух направлений. Для управления пересылкой информации предназначены сигналы квитирования. Процесс квитирования аналогичен описанному в разделе 4.5.2, но в данном случае роль хозяина шины выполняет целевой контроллер. Сигналы –REQ и –ACK соответствуют сигналам Master-ready и Slave-ready на рис. 4.26 и 4.27. Для выполнения операции ввода (пересылки данных от целевого контроллера к инициатору) целевой контроллер активизирует сигнал –I/O на все время этой операции. Кроме того, он может активизировать сигнал –C/D, указывающий, что пересылаются не данные, а команда или информация о состоянии.

В высокоскоростных версиях шины SCSI используется технология, называемая тактированием двумя фронтами или двойным переходом (Double transition, DT). На рис. 4.26 и 4.27 для выполнения каждой операции пересылки данных необходимо, чтобы для каждого сигнала квитирования был выполнен переход от высокого уровня сигнала к низкому, а затем переход от низкого к высокому уровню. Тактирование двумя фронтами означает, что данные пересылаются и на переднем, и на заднем фронте сигнала, в результате чего скорость пересылки удваивается.

В конце фазы пересылки целевой контроллер снимает сигнал –BSY и тем самым освобождает шину для использования другими устройствами. Позднее, когда будут готовы новые данные, он может возобновить соединение с инициатором. Для этого используется описанная ниже операция повторного выбора.

Повторный выбор

Когда логическое соединение временно разорвано и целевой контроллер готов его восстановить, он сначала должен снова получить управление шиной. Для этого он начинает цикл арбитража и, выиграв таковой, выбирает инициирующий контроллер — точно так же, как в свое время инициирующий контролер выбрал

его. Только теперь сигнал –BSY, сообщающий о возобновлении соединения, активизирует инициатор. Прежде чем начать пересылку данных, инициатор должен передать управление шиной целевому контроллеру. Для этого после выбора инициатора целевой контроллер, в свою очередь, активизирует сигнал –BSY. Тем временем инициатор выдерживает небольшую паузу, чтобы целевой контроллер успел активизировать сигнал –BSY, а затем снимает этот сигнал. В результате соединение возобновлено и целевой контроллер управляет шиной, что необходимо для выполнения пересылки данных.

Описанная выше сигнальная схема шины определяет механизм установки логического соединения между контроллерами и обмена сообщениями. В любой момент соединение может быть временно прервано и снова восстановлено. Структура и содержимое различных типов пакетов, которыми контроллеры обмениваются в разных ситуациях, определяются стандартом SCSI. Инициатор использует эти пакеты для отправки целевому контроллеру полученных от процессора команд. В ответ целевой контроллер присылает ему информацию о состоянии и выполняет операции пересылки данных. Такие операции управляются целевым контроллером, поскольку только он знает, когда будут готовы данные, а следовательно, когда необходимо временно разорвать и снова восстановить соединение.

Дополнительную информацию о шине SCSI и различных SCSI-устройствах вы найдете на web-узле комитета по стандартизации, расположенному по адресу www.ansi.org.

4.7.3. Шина USB

Объединение компьютеров и коммуникационных технологий привело к настоящей информационной революции. Современные компьютерные системы включают множество устройств, таких как клавиатуры, микрофоны, цифровые видеокамеры, динамики, дисплеи. И почти все они имеют проводное или беспроводное соединение с Интернетом. Одним из важнейших требований к таким системам является наличие простого и недорогого механизма подключения к компьютеру внешних устройств. Одной из последних разработок в этой области стала универсальная последовательная шина — Universal Serial Bus (USB). USB является промышленным стандартом, разработанным объединенными усилиями ряда компьютерных и коммерческих компаний, к числу которых относятся Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, Philips.

USB поддерживает два режима функционирования, получивших названия низкоскоростной (1,5 Мбит/с) и полноскоростной (12 Мбит/с). В последней версии этой спецификации, USB 2.0, введен третий режим, названный высокоскоростным (480 Мбит/с). Шина USB быстро завоевывает популярность на рынке и с появлением высокоскоростного режима может стать основным средством взаимодействия большинства компьютерных устройств.

Разработчики USB ставили перед собой следующие задачи:

- создать простую, дешевую и удобную систему соединения, позволяющую преодолевать сложности, возникающие из-за ограниченного числа имеющихся в компьютерах портов ввода-вывода;

- учесть широкий диапазон параметров пересылаемых данных, присущих различным устройствам ввода-вывода, в том числе модемам;
- облегчить для пользователей процесс подключения устройств за счет поддержки режима plug-and-play.

Далее перечисленные задачи будут рассмотрены более подробно, а затем мы перейдем к обсуждению технических аспектов технологии USB.

Ограниченное количество портов

Параллельный и последовательный порты, о которых рассказывалось в разделе 4.6, имеют универсальные разъемы, через которые к компьютеру можно подключать самые разнообразные низко- и среднескоростные устройства. Однако по практическим соображениям типичный компьютер имеет всего несколько таких портов. Для добавления нового порта пользователь должен открыть корпус компьютера, чтобы получить доступ к внутренней шине расширения и вставить в ее разъем новую интерфейсную плату. Кроме того, пользователь должен знать, как настроить новое устройство и его программное обеспечение. Шина USB обеспечивает возможность подключения к компьютеру большего количества устройств в любой момент и без необходимости открывать его корпус.

Характеристики устройств

Типы устройств, которые можно подключать к компьютеру, довольно разнообразны. Скорость, объемы и временные характеристики процесса обмена данными с такими устройствами тоже бывают очень разными.

Например, клавиатура генерирует один символ каждый раз, когда нажимается какая-либо клавиша, а это может произойти в любой момент. При этом данные немедленно должны быть переданы в компьютер. Поскольку событие нажатия клавиши не синхронизировано ни с одним другим событием в компьютерной системе, генерируемые клавиатурой данные называются *асинхронными*. Более того, скорость, с которой они генерируются, очень мала. Она ограничена скоростью работы оператора, который может вводить около 100 байт (символов) в секунду, что составляет менее 1000 бит в секунду.

Существует множество других устройств, которые генерируют данные такого типа. К их числу относятся мыши и игровые манипуляторы.

Рассмотрим другой источник данных. Многие компьютеры снабжены микрофонами — либо встроенными, либо подключенными в качестве внешних устройств. Воспринимаемый микрофоном звук преобразуется в аналоговый электрический сигнал, который перед обработкой компьютером должен быть преобразован в цифровую форму. Для этого аналоговый сигнал дискретизируется и аналогово-цифровой преобразователь (АЦП) через определенные промежутки времени (период дискретизации) замеряет характеристики звука и генерирует соответствующие n-разрядные значения. Аналоговый звуковой сигнал, замеряемый в конкретный момент времени, называется отсчетом. Количество битов, n, выбирается исходя из требуемой точности представления звука. Позже, когда эти данные передаются на динамик, обратный цифро-аналоговый преобразователь (ЦАП) восстанавливает исходный аналоговый сигнал из цифрового формата.

В процессе дискретизации генерируется непрерывный поток числовых значений, поступающих через равные промежутки времени. Этот процесс синхронизируется с помощью тактового сигнала. Процесс, последовательные события которого происходят через равные промежутки времени, называется *изохронным*.

Для того чтобы при оцифровке звука сохранялись те его составляющие, которые меняются с очень большой скоростью, частота дискретизации должна быть очень высокой. Если она составляет s значений в секунду, максимальная частота звука, фиксируемая в ходе оцифровки, будет равна $s/2$. Например, человеческая речь адекватно записывается при частоте дискретизации 8 кГц, что позволяет сохранять звуковые частоты до 4 кГц. Для более высоких звуков, которые должны записываться музыкальными системами, требуется большая частота дискретизации. Стандартная частота дискретизации цифрового звука составляет 44,1 кГц. Каждый отсчет передается посредством 4 байт данных, что позволяет представить достаточно широкий диапазон уровней звука (динамический диапазон), необходимый для высококачественного воспроизведения звука. В результате скорость потока данных составляет около 1,4 Мбит/с.

Для процесса оцифровки голоса или музыки важно сохранить точное соответствие частоты дискретизации записи и воспроизведения звука. Значительное расхождение тактовой частоты при записи и воспроизведении звука совершенно недопустимо. Поэтому механизм пересылки данных между компьютером и музыкальной системой должен обеспечивать строго одинаковые промежутки времени между отсчетами. В противном случае потребуется сложная схема буферизации и повторного тактирования. С другой стороны, вполне допустимы отдельные ошибки и пропущенные отсчеты. Они либо вовсе не замечаются слушателем, либо приводят к возникновению щелчов при воспроизведении записи.

К качеству графических и видеофайлов предъявляются похожие требования, но для их передачи нужна значительно более широкая полоса пропускания канала. Термин «полоса пропускания» означает пропускную способность коммуникационного канала, измеряемую в битах или байтах в секунду. Для воспроизведения видео с качеством коммерческого телевидения составляющие его отдельные изображения должны иметь объем около 16 Кбайт и передаваться со скоростью 30 изображений в секунду, для чего необходима полоса пропускания 44 Мбит/с. Для более качественного видео, такого как HDTV (High Definition TV), требуются более высокие характеристики.

Запоминающие устройства большого объема, в частности жесткие диски и CD-ROM, предъявляют несколько иные требования к коммуникационным интерфейсам. Как рассказывается в главе 5, эти устройства являются частью иерархии памяти компьютера. Их соединение с компьютером должно обладать пропускной способностью порядка 50 Мбит/с. Особенности функционирования диска таковы, что при его работе происходят задержки порядка миллисекунды. Поэтому небольшие дополнительные задержки при пересылке данных между диском и компьютером не имеют значения, равно как и незначительные колебания частоты.

Технология plug-and-play

По мере того как компьютеры все активнее входят в нашу повседневную жизнь, их существование становится все более незаметным (как говорят в компьютерном

мире, прозрачным) для пользователя. Например, используя домашний театр, в состав которого входит как минимум один компьютер, пользователь не должен выключать этот компьютер или перезапускать систему, для того чтобы присоединить или отсоединить одно из устройств.

Технология plug-and-play предполагает, что новое устройство, скажем дополнительный динамик, можно подключить в любое время прямо к работающей системе. Система должна сама обнаружить новое устройство, идентифицировать его и соответствующее ему программное обеспечение (драйвер, а также другие необходимые для его работы средства), назначить ему адреса и установить логические соединения для взаимодействия с другими устройствами.

Этот принцип налагает немалое количество требований и имеет множество следствий на всех уровнях системы, от аппаратного обеспечения до операционной системы и прикладных программ. Одной из главных задач разработчиков шины USB была именно реализация принципа plug-and-play.

Архитектура USB

Из сказанного выше следует, что для поддержки таких разнообразных устройств необходима коммуникационная система, отличающаяся низкой стоимостью, гибкостью и широкой полосой пропускания для пересылки данных. При этом нужно учесть, что устройства ввода-вывода могут располагаться на некотором расстоянии от компьютера. Для удовлетворения высоких требований к полосе пропускания требуется широкая шина, по которой могут параллельно передаваться 8, 16 и более битов информации. Однако большое количество проводов повышает стоимость и усложняет конструкцию шины, делает ее неудобной для использования. Кроме того, из-за проблем со сдвигом данных (об этом говорилось в разделе 4.5.3) довольно сложно разработать широкую шину, по которой можно было бы передавать данные на большие расстояния. С расстоянием величина сдвига увеличивается, из-за чего снижается скорость пересылки данных.

Для шины USB выбран последовательный формат пересылки данных, обеспечивающий ее наименьшую стоимость и наибольшую гибкость. Тактирующий сигнал и данные кодируются вместе и передаются как единый сигнал. В результате нет никаких ограничений в отношении тактовой частоты или расстояний, связанных со сдвигом данных, благодаря чему становится возможной высокая пропускная способность соединений с высокой тактовой частотой. Как уже упоминалось ранее, шина USB поддерживает три скорости пересылки данных, а именно 1,5; 12 и 480 Мбит/с, что соответствует нуждам самых разных устройств ввода-вывода.

Для того чтобы к шине USB можно было одновременно подключать большое количество устройств, удаляемых и подсоединяемых в любое время, эта шина имеет древовидную структуру (рис. 4.43). В узлах дерева располагаются устройства, называемые *хабами* и действующие как промежуточные управляющие компоненты между хостом и устройствами ввода-вывода. *Корневой хаб* соединяет все дерево с хост-компьютером. Листьями дерева являются устройства ввода-вывода (клавиатура, динамики, соединение с Интернетом, цифровой телевизор и т. п.), в терминологии USB называемые *функциями*. Для согласованности с остальной частью книги мы будем по-прежнему называть их устройствами ввода-вывода.

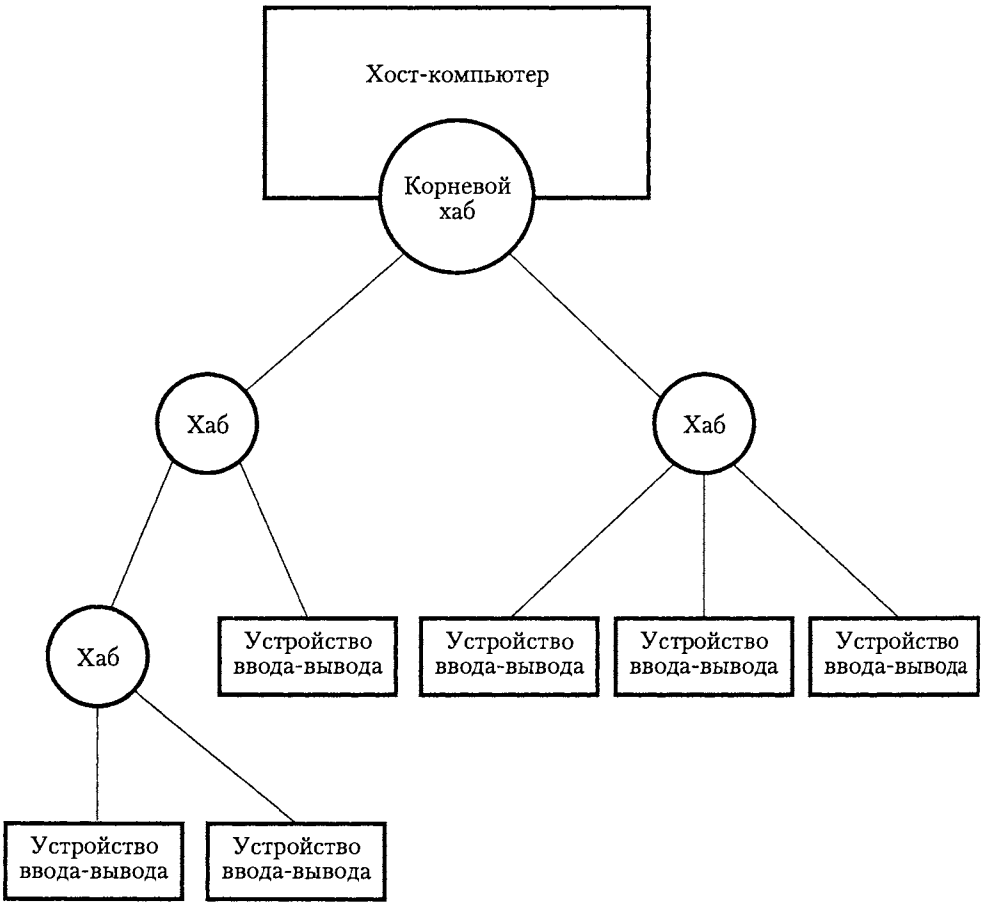


Рис. 4.43. Структура дерева USB

Показанная на рисунке древовидная структура позволяет соединять множество устройств с помощью простых последовательных соединений «точка-точка». Каждый хаб имеет ряд портов, к которым можно подключать любые устройства, в том числе и другие хабы. В нормальном режиме хаб копирует полученное входное сообщение в свои выходные порты. В результате посланное компьютером сообщение передается всем устройствам ввода-вывода, но отвечает на него только адресуемое устройство. В этом отношении USB функционирует подобно шине, показанной на рис. 4.1. Однако в отличие от этой шины сообщение от устройства ввода-вывода пересылается только вверх, в направлении корневого узла, и другие устройства его не получают. Таким образом, USB дает возможность хосту взаимодействовать с устройствами ввода-вывода, но не позволяет им взаимодействовать друг с другом.

Давайте посмотрим, как древовидная структура соединений USB отвечает задачам этой шины. Дерево позволяет подключать к компьютеру большое количество

устройств, используя всего один или несколько портов (корневые хабы). В то же время каждое устройство подключается к компьютеру или хабу с помощью последовательного соединения «точка-точка». Как будет показано ниже, это обстоятельство играет очень важную роль в реализации принципа plug-and-play. Кроме того, из-за некоторых особенностей электрического характера последовательная пересылка данных по таким шинам, как показано на рис. 4.1, выполняется гораздо проще параллельной пересылки. И при этом данные могут пересылаться гораздо быстрее, а кабели могут быть намного длиннее.

В основе функционирования шины USB лежит принцип опроса устройств. Устройство может отослать сообщение только в ответ на запрос хоста. Поэтому передаваемые хосту сообщения не конфликтуют и не пересекаются друг с другом, и никакие два устройства не могут отослать сообщения одновременно. Это ограничение позволяет применять простые и недорогие хабы.

Описанный выше режим функционирования пригоден для всех устройств, работающих на низкой или полной скорости. Однако с появлением высокоскоростной шины USB версии 2.0 связано одно исключение. Рассмотрим ситуацию, показанную на рис. 4.44. Хаб A подключен к корневому хабу через высокоскоростное соединение. Он обслуживает одно высокоскоростное устройство C и одно низкоскоростное устройство D. Как правило, сообщения устройству D от корневого хаба пересылаются на низкой скорости. Но при скорости 1,5 Мбит/с даже для пересылки короткого сообщения понадобится несколько десятков миллисекунд. И все это время никакие другие данные пересылаться не могут, из-за чего снижается эффективность высокоскоростного соединения и происходят задержки, неприемлемые для высокоскоростных устройств. Для разрешения данной проблемы протокол USB требует, чтобы сообщения, передаваемые через высокоскоростное соединение, всегда пересылались на высокой скорости, даже если их конечным получателем является низкоскоростное устройство. Таким образом, сообщение, предназначенное для устройства D, пересылается от корневого хаба к хабу A на высокой скорости, а дальше, к устройству D, — на низкой. Последний этап пересылки займет много времени, в течение которого разрешается производить высокоскоростной трафик к другим узлам. Например, пока низкоскоростное сообщение проходит путь от хаба A к устройству D, корневой хаб может обменяться несколькими сообщениями с устройством C. В течение этого времени шина делится между высокоскоростным и низкоскоростным трафиком. Сообщение, следующее к устройству D, предваряется и завершается специальными командами для хаба A перейти в режим разделения трафика, а затем выйти из такового.

Стандартом USB определяются особенности аппаратной реализации соединений USB, а также структура программного обеспечения хоста и требования к самому обеспечению. Последнее предназначено для поддержки двунаправленных коммуникационных соединений между прикладным программным обеспечением и устройствами ввода-вывода. Эти соединения называются *каналами* (pipe). Любые данные, входящие в канал с одного конца, обязательно достигают другого. Все вопросы, связанные с адресацией, тактированием, выявлением ошибок и восстановлением решаются посредством протоколов USB.

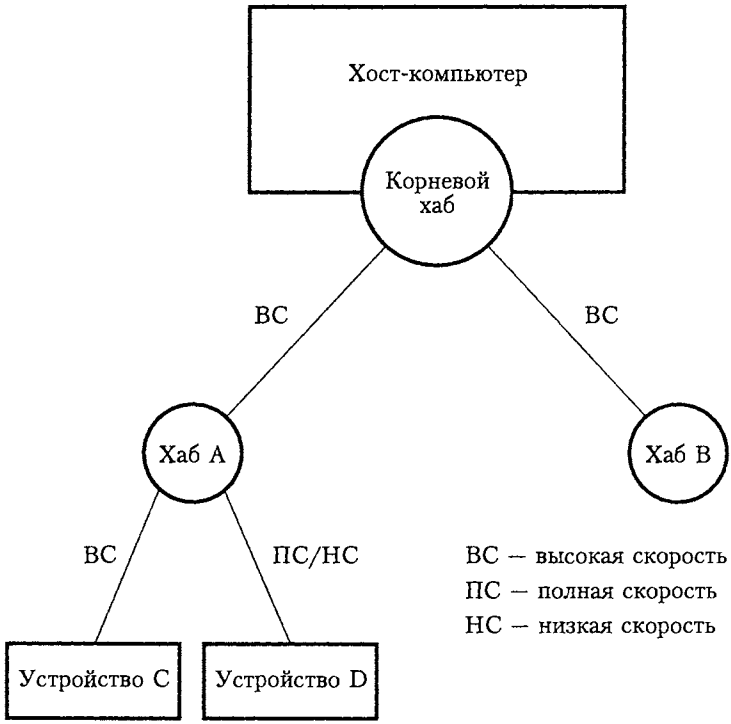


Рис. 4.44. Разделение шины

Мы уже упоминали в разделе 4.2.6, что программное обеспечение, пересылающее данные от устройства к устройству ввода-вывода, называется драйвером этого устройства. Каждый драйвер разрабатывается с учетом характеристик того конкретного устройства, для которого он предназначен. Поэтому более точное определение канала USB таково: соединение устройства ввода-вывода с его драйвером. Это соединение устанавливается после подключения устройства и назначения ему программным обеспечением USB уникального адреса. При наличии соединения данные могут пересылаться по каналу по мере надобности.

Далее будут рассмотрены принципы адресации устройств, подключаемых к шине USB, после чего мы перейдем к описанию различных способов пересылки данных по шине.

Адресация

Когда в начале этой главы речь шла об операциях ввода-вывода, мы говорили, что устройства ввода-вывода обычно идентифицируются с помощью назначаемых им уникальных адресов памяти. Одно устройство может содержать несколько адресуемых запоминающих элементов, позволяющих программному обеспечению направлять ему и получать от него управляющую информацию, информацию о состоянии и данные. Корневой хаб USB соединяется с шиной процессора, который обращается к хабу как к единственному устройству. Программное обеспечение

хост-компьютера взаимодействует с отдельными устройствами, подключенными к шине USB, направляя им пакеты информации, которую корневой хаб перенаправляет по цепочке соединений USB конкретному устройству.

Каждому устройству на шине USB, будь то хаб или устройство ввода-вывода, назначается 7-битовый адрес. Этот адрес локален для дерева USB и никак не соотносится с адресами, используемыми на шине процессора. К хабу может быть подключено любое количество устройств и других хабов, адреса которых назначаются произвольным образом. Когда устройство активизируется или подключается к хабу, оно имеет адрес 0. Аппаратное обеспечение хаба обнаруживает новое устройство, о чем делает соответствующую пометку в своей информации о состоянии. Периодически хост опрашивает все хабы, собирая сведения об их состоянии, и узнает о добавленных или удаленных устройствах. Когда хост узнает о подключении нового устройства, он с помощью специальной последовательности команд направляет в порт хаба сигнал сброса, считывает из памяти устройства информацию о его возможностях, направляет этому устройству конфигурационную информацию и присваивает ему уникальный USB-адрес. После этого начинается обычное функционирование устройства, которое теперь имеет новый адрес.

Описанная процедура инициализации подключения является основой реализации принципа plug-and-play. Ею всецело управляет программное обеспечение хоста. Оно имеет возможность узнать о подключении устройства, прочитать информацию об этом устройстве (которая обычно хранится в его встроенной памяти небольшого объема, доступной только для чтения), направить ему конфигурационные команды, необходимые для настройки параметров, и, наконец, присвоить ему уникальный USB-адрес. Единственное, что при этом требуется от пользователя, — это подключить устройство к порту хаба и включить его питание.

При выключении устройства выполняются аналогичные действия. Соответствующий хаб сообщает об этом факте программному обеспечению USB, которое обновляет свои таблицы. Если отключенное устройство само является хабом, программное обеспечение, конечно же, логически отсоединяет и все подключенные к нему устройства. Программное обеспечение USB должно постоянно иметь полную картину топологии шины и подключенных к нему устройств.

Такие места хранения информации, как регистры состояния, управления и данных, называются *конечными точками*. Они идентифицируются 4-разрядными числами. Собственно говоря, 4-разрядное значение идентифицирует пару конечных точек: одну для ввода данных, а другую — для их вывода. Таким образом, у устройства может быть до 16 входных-выходных пар конечных точек. Каждый канал USB, будучи двунаправленным, соединяется с одной такой парой. Один канал, под номером 0, соединенный с конечными точками, существует всегда и создается сразу после включения или перезапуска устройства. Это управляющий канал, который используется программным обеспечением USB в процессе инициализации устройства. В ходе этого процесса создаются другие каналы, количество которых зависит от потребностей и сложности устройства. Как рассказывается далее, 4-разрядный номер конечной точки является частью адресной информации, которую хост отправляет устройствам ввода-вывода.

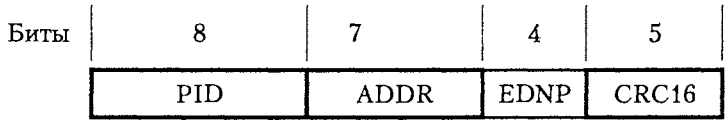
Протоколы USB

Информация, пересылаемая через соединения USB, организуется в пакеты, каждый из которых включает один или несколько байтов данных. Существует множество типов пакетов, выполняющих разные управляющие функции. Рассматривая работу шины USB на примерах, мы расскажем о нескольких важнейших типах пакетов и покажем, как они используются.

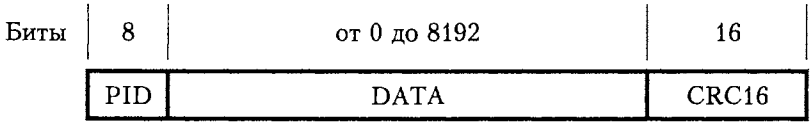
Пересылаемую по шине USB информацию можно разделить на две категории: управляющая информация и данные. Управляющие пакеты используются для адресации устройств при инициировании пересылки данных, а также для подтверждения факта получения правильных данных и сообщений об ошибках. Пакеты данных содержат входные и выходные данные, которыми хост обменивается с устройством, и некоторую другую информацию.

Каждый пакет состоит из одного или нескольких полей, содержащих разные типы информации. Первое поле любого пакета называется идентификатором и обозначается как PID. Оно идентифицирует тип пакета. В этом поле четыре бита информации, которые передаются дважды. В первый раз пересылаются их реальные значения, а во второй — дополненные, как показано на рис. 4.45, *а*. Это позволяет устройству-получателю проверить достоверность полученного байта PID.

а



б



в

Рис. 4.45. Форматы пакетов USB: поле идентификатора пакета (*а*); пакет маркера, IN или OUT (*б*); пакет данных (*в*)

Четыре бита PID идентифицируют один из 16 возможных типов пакетов. Некоторые управляющие пакеты, и в частности ACK (Acknowledge — подтверждение), состоят только из байта PID. Пакеты, используемые для управления операциями пересылки данных, называются пакетами маркера. Их формат проиллюстрирован

на рис. 4.45, *б*. Пакет маркера начинается с поля PID, в котором содержится одно из двух значений, идентифицирующих тип пакета, — IN или OUT. Пакеты типа IN предназначены для управления операциями ввода, а пакеты типа OUT — операциями вывода. За полем PID следует 7-разрядный адрес устройства и 4-разрядный номер конечной точки внутри этого устройства. Последние пять разрядов в пакете предназначены для его же проверки на наличие ошибок. Для этой цели предназначен метод, называемый циклическим контролем избыточности (Cyclic Redundancy Check, CRC). Биты CRC вычисляются на основе содержимого полей адреса и конечной точки. С помощью обратного вычисления устройство-получатель может определить наличие ошибок в пакете.

Пакеты, содержащие входные и выходные данные, имеют формат, показанный на рис. 4.45, *в*. За полем идентификатора пакета следуют не более 8192 бит данных, затем 16 контрольных битов. Для идентификации пакетов используются три разных значения PID, так что пакеты данных могут иметь номера 0, 1 и 2. Обратите внимание, что пакеты данных не содержат ни адреса устройства, ни номера конечной точки. Такого рода информация включается в пакет IN или OUT, инициировавший пересылку данных.

Рассмотрим пример, в котором выходное устройство подключено к хабу USB, который, в свою очередь, подключен к хост-компьютеру. Пример операции вывода для такой конфигурации показан на рис. 4.46. Хост-компьютер отсылает хабу пакет маркера типа OUT, а за ним пакет данных, содержащий выходную информацию. Поле PID пакета данных идентифицирует его как пакет данных с номером 0. Хаб убеждается, что пересылка выполнена без ошибок, проверив контрольные биты пакета, а затем отсылает хосту пакет-подтверждение (ACK). Пакет маркера и пакет данных хаб отсылает далее, вниз по дереву. Эти пакеты получают все устройства ввода-вывода, но только одно из них распознает свой адрес в пакете маркера и принимает данные, содержащиеся в следующем пакете. Убедившись, что переданная информация не содержит ошибок, устройство отсылает хабу пакет ACK.

Пакет данных, успешно пересланный на полной или низкой скорости, имеет соответственно номер 0 или 1. Это упрощает процесс восстановления данных после обнаружения ошибок, связанных с пересылкой. Если в результате ошибки, допущенной при пересылке, теряется пакет маркера, пакет данных или пакет-подтверждение, отправитель должен повторно отослать весь набор пакетов. Проверив номер пакета данных в поле PID, получатель может выявить и проигнорировать повторяющиеся пакеты. Высокоскоростные пакеты данных последовательно нумеруются как 0, 1, 2, 0 и т. д.

Операции ввода выполняются аналогичным образом. Хост отсылает пакет маркера типа IN, содержащий адрес устройства. Этот пакет используется для опроса и дает указание распознавшему свой адрес устройству отправить имеющиеся у него входные данные. Устройство отвечает отправкой пакета данных, а также пакета ACK. Если устройство не содержит готовых к отправке данных, оно сообщает об этом, отсылая пакет NAK.

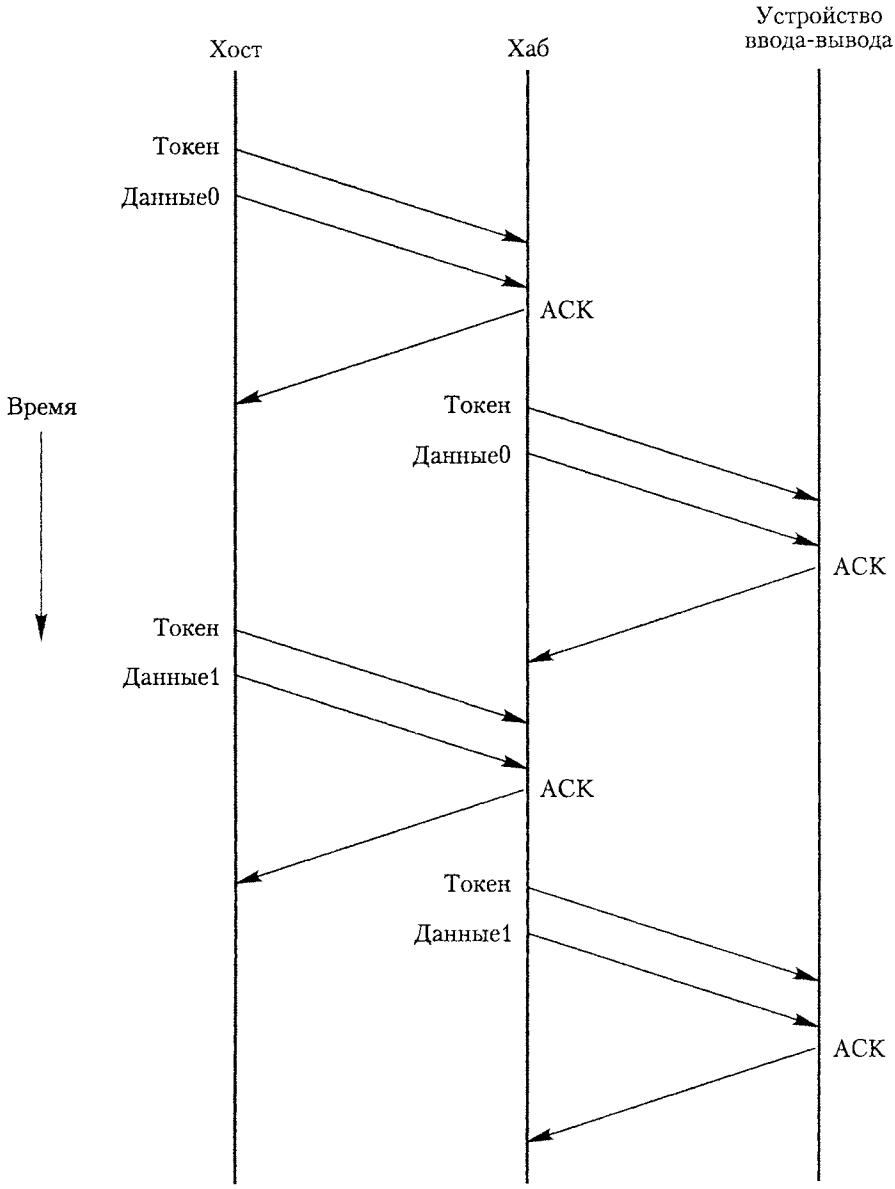


Рис. 4.46. Пересылка выходных данных

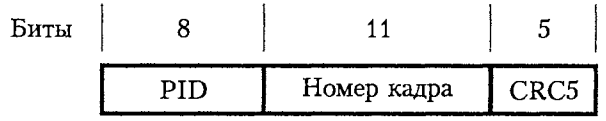
Мы уже говорили о том, что если к шине наряду с низко- и полноскоростными линиями подключены высокоскоростные, шина может работать в разделенном режиме, чтобы не задерживать сообщения, направляемые по высокоскоростным линиям. В таких случаях пакеты IN и OUT, предназначенные для полно- и низкоскоростных устройств, предваряются специальными управляющими пакетами, инициирующими переход в режим разделенного трафика.

У читателя должно уже сложиться достаточно четкое представление о сути протоколов, используемых для пересылки данных по шине USB. Таких протоколов существует очень много, не меньше чем способов выполнения транзакций для разных устройств. Их подробное описание вы найдете по адресу www.usb.org/developers, в документах, составляющих спецификацию Universal Serial Bus Specification.

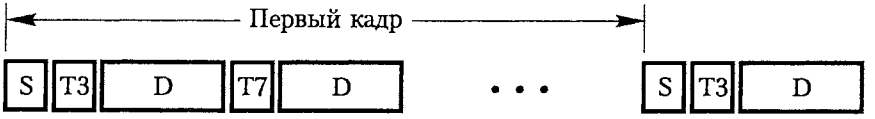
Изохронный трафик по шине USB

Одной из ключевых задач USB является поддержка пересылки изохронных данных, например оцифрованного голоса. Устройствам, генерирующим и получающим изохронные данные, для управления процессом дискретизации и восстановления сигнала необходима тактовая информация. С этой целью передача данных по шине USB осуществляется покадрово. В случае полно- и низкоскоростных данных длительность кадра составляет 1 с. Ровно через каждую миллисекунду корневой хаб генерирует пакет SOF (Start Of Frame — начало кадра), отмечающий начало кадра.

Момент прибытия пакета SOF устройство может воспринимать как тактовый сигнал. Для того чтобы упростить функционирование устройств, которым требуются более длительные такты, в пакет SOF включается 11-разрядный номер кадра (рис. 4.47, *а*). Вслед за отправкой пакетов SOF хост выполняет пересылку входных данных изохронным устройствам. Это означает, что каждое из подключенных к шине изохронных устройств сможет каждую миллисекунду получать или отправлять данные.



а



S — пакет начала кадра SOF
T*n* — пакет маркера, где *n* является адресом
D — пакет данных
A — пакет подтверждения ACK

б

Рис. 4.47. Пакеты шины USB: пакет SOF (*а*); структура кадра (*б*)

Основным требованием к изохронному трафику является строгое тактирование. При этом вполне допустимы отдельные случайные ошибки. Это означает, что нет нужды ни в повторной отправке потерянных пакетов, ни в подтверждении факта получения каждого пакета. На рис. 4.47, б показаны два первых фрагмента данных, следующих за пакетом SOF, а именно управляющий пакет с адресом устройства 3 и пакет данных для этого устройства. Данные могут быть входными или выходными, в зависимости от того, относится управляющий пакет к типу IN или OUT. Подтверждающий пакет в этом случае не отправляется. Следующая группа пакетов относится к устройству 7.

Пакет данных для устройства 3 может, к примеру, содержать 8 байт данных. В каждом кадре отправляется один такой пакет, в результате чего создается изохронный канал с пропускной способностью 64 Кбит/с. Такой канал может использоваться для голосовой связи. Для пересылки 8 байт данных необходим 3-байтовый пакет маркера, за которым следует 11-байтовый пакет данных (включающий поля PID и CRC), — итого 132 бита. Еще как минимум 3 байта нужны для синхронизационного пакета и пакета, отмечающего конец последовательности пакетов. При скорости 12 Мбит/с вся пересылка займет около 13 мкс. Очевидно, что в течение оставшегося времени кадра можно обслужить еще несколько таких устройств. После обслуживания всех подключенных к шине изохронных устройств оставшееся время кадра используется для обслуживания асинхронных устройств и обмена управляющей информацией и информацией о состоянии.

Изохронные данные могут передаваться только по полно- и высокоскоростным линиям. В случае высокоскоростных соединений в течение каждого кадра, длительность которого составляет 1 мс, пакет SOF повторяется восемь раз, через равные промежутки времени, в результате чего создаются микрокадры длительностью 125 мкс.

Электрические характеристики шины

Кабели, используемые для USB-соединений, состоят из четырех проводов. Два из них (+5 В и «земля») предназначены для подачи питания. Поэтому хабы и устройства ввода-вывода могут иметь собственные источники питания, а могут питаться непосредственно от шины. Два других провода предназначены для пересылки данных. В случае низкоскоростных устройств передача единиц и нулей производится путем установки на одном или двух сигнальных проводах напряжения 5 В. При высоковольтном соединении применяется дифференциальная пересылка.

4.8. Резюме

В настоящей главе были рассмотрены три базовых подхода к операции пересылки входных и выходных данных. Простейшим из них является программируемый ввод-вывод, который управляется процессором, выполняющим команды программы. Второй подход основан на использовании прерываний; этот механизм позволяет прерывать нормальное выполнение программы для обслуживания запросов с более высоким приоритетом, которым срочно требуется уделить внимание. Механизмы для обработки подобных ситуаций предусмотрены во всех компьютерах,

но схемы обработки прерываний у одних компьютеров проще, а у других, наоборот, сложнее. Третья схема ввода-вывода основана на прямом доступе к памяти — аппаратной технологии, суть которой заключается в том, что специальный контроллер ПДП пересылает данные между устройством ввода-вывода и основной памятью без постоянного участия процессора. При этом доступ к памяти по очереди получают то процессор, то контроллер ПДП — каждый для своих нужд.

Были рассмотрены три популярных стандарта подключения к компьютеру устройств ввода-вывода: PCI, SCSI и USB. Они представляют различные подходы к передаче данных для устройств различных типов, поддерживающих механизм plug-and-play.

Упражнения

4.1. Бит состояния в интерфейсной схеме устройства ввода очищается сразу после чтения данных из входного буфера данных. Почему?

4.2. Напишите программу, которая выводила бы на видеодисплей содержимое 10 байт основной памяти в шестнадцатеричном формате. Можете использовать либо ассемблерные команды выбранного вами процессора, либо псевдокоманды. Начните с адреса памяти LOC и используйте для кодировки каждого байта по два шестнадцатеричных символа. Содержимое последовательных байтов должно быть разделено пробелом.

4.3. Адресная шина компьютера состоит из 16 адресных линий, A_{15-0}. Если некоторому устройству присвоен адрес $7CA4_{16}$, а декодер адреса для этого устройства игнорирует линии A_8 и A_9, то на какие адреса будет отвечать данное устройство?

4.4. В чем различие между подпрограммой и программой обработки прерывания?

4.5. В этой главе предполагалось, что прерывания не подтверждаются до тех пор, пока не будет закончено выполнение текущей машинной команды. Рассмотрите возможность приостановления операции прямо в тот момент, когда команда будет выполнена наполовину. Какие трудности могут при этом возникнуть?

4.6. К шине компьютера подключены три устройства: A, B и C. При выполнении операций ввода-вывода всеми этими устройствами используются прерывания. Вложенность прерываний для устройств A и B не допускается, но запросы прерывания C могут обслуживаться во время обработки прерываний A и B. Предложите различные способы выполнения этой задачи для каждого из перечисленных ниже случаев.

   а) Компьютер имеет одну линию запроса прерывания.

   б) Имеются две линии запроса прерывания, INTR1 и INTR2, первая из которых обладает более высоким приоритетом.

   Объясните, когда и как разрешаются и запрещаются прерывания в каждом из этих случаев.

4.7. Рассмотрите работу компьютера, в котором несколько устройств соединены с общей линией запроса прерывания, как на рис. 4.8, *а*. Объясните, как добиться того, чтобы прерывания от устройства *j* принимались до завершения выполнения программы обработки прерываний от устройства *i*. Укажите, в какие моменты должны разрешаться и запрещаться прерывания в разных точках системы.

4.8. Обратимся к гирляндной схеме, приведенной на рис. 4.8, *а*. Предположим, сгенерировав запрос прерывания, устройство дожидается подтверждения и, получив таковое, тут же снимает запрос. Обязательно ли перед входом в программу обработки прерывания следует запрещать прерывания в процессоре? Почему?

4.9. Последовательные блоки данных длиной N байт каждый должны считываться из символьного устройства ввода, и для каждого из них программа PROG должна выполнять некоторые вычисления. Напишите управляющую программу CONTROL для процессоров 68000, ARM и Pentium, выполняющую следующие функции:

а) чтение блока данных 1;

б) активизацию программы PROG и установку в ней ссылки на блок 1 в основной памяти;

в) чтение блока 2 с использованием прерываний, пока программа PROG выполняет вычисления для блока 1;

г) переход программы PROG к блоку 2 и параллельное чтение блока 3.

Заметьте, что программа CONTROL должна поддерживать корректные указатели на буферы, считать символы и передавать управление программе PROG, которая может выполняться быстрее или, наоборот, дольше, чем ввод блока данных.

4.10. Компьютер должен принимать символы от 20 видеотерминалов. Для хранения данных каждого терминала в основной памяти выделяется область, на которую указывает указатель PNTRn, где n — значение из диапазона от 1 до 20. Сбор входных данных от терминалов должен производиться параллельно с выполнением программы PROG. Это можно сделать одним из двух способов.

а) Каждые T с программа PROG вызывает опрашивающую подпрограмму POLL. Та по очереди проверяет состояние каждого из 20 терминалов и пересылает в память введенные данные. Затем она возвращает управление программе PROG.

б) Когда в интерфейсном буфере любого из терминалов готов очередной символ, генерируется запрос прерывания. В ответ на это выполняется программа обработки прерывания INTERRUPT. После опроса регистров состояния программа INTERRUPT пересылает введенный символ и возвращает управление программе PROG.

Напишите программы POLL и INTERRUPT, используя либо псевдокод, либо язык ассемблера любого из процессоров. Пусть максимальная скорость ввода символов для любого из терминалов составляет c символов

в секунду, а средняя скорость — rc, где $r \leq 1$. Если будет использован первый метод, то каково максимальное значение T, при котором можно гарантировать, что ни один из введенных символов не будет потерян? Каким это значение будет в случае применения второго метода? Оцените среднее время (в процентах), затрачиваемое на обслуживание терминалов по каждому из методов при условии, что $c = 100$ символов в секунду, а $r = 0{,}01$; 0,1; 0,5 и 1. Предполагается, что на опрос 20 устройств программа POLL затрачивает 800 нс, а на обработку прерывания от устройства уходит 200 нс.

4.11. Имеется устройство ввода-вывода, использующее функцию векторных прерываний процессора 68000.

а) Опишите последовательность шагов, выполняемых при получении процессором запроса прерывания, и определите количество операций пересылки по шине, необходимых для каждого из этих шагов. При этом учитывать подробности работы шины и микропрограмм не следует.

б) Получив запрос прерывания, процессор завершает выполнение текущей команды и только после этого принимает запрос. Проанализируйте таблицу команд в приложении В и оцените максимальное количество операций пересылки данных в память и из памяти, выполняемых в течение этого времени.

в) Оцените количество необходимых для выполнения первой команды программы обработки прерывания операций пересылки данных по шине, выполняемых с момента запроса устройством прерывания и до выборки информации из памяти.

4.12. Логическая схема, необходимая для реализации системы приоритетов, показана на рис. 4.8, *б*. Система включает три линии запроса прерываний. Когда получен запрос по линии INTR*i*, система генерирует подтверждение на линии INTA*i*. Если получено более одного запроса прерывания, подтверждается только запрос с наивысшим приоритетом. Порядок приоритетов таков:

приоритет INTR1 > приоритета INTR2 > приоритета INTR3

а) Постройте таблицу истинности для каждого из выходов INTA1, INTA2 и INTA3.

б) Приведите логическую схему для реализации этой системы приоритетов.

в) Можно ли данную схему без труда расширить с целью ввода большего количества линий запроса прерывания?

г) Путем добавления входов DECIDE и RESET модифицируйте свою схему таким образом, чтобы линия INTA*i* устанавливалась в 1 в ответ на получение сигнала на входе DECIDE и сбрасывалась в 0 в ответ на получение сигнала на входе RESET.

4.13. Для прерываний и арбитража необходим механизм выбора одного из нескольких запросов на основе их приоритетов. Разработайте схему, реализующую циклическую систему приоритетов для четырех входных линий,

от REQ1 до REQ4. Первоначально линия REQ1 имеет наивысший приоритет, а линия REQ4 — самый низкий. После обслуживания запроса по одной из линий эта линия получает наинизший приоритет, а остальные линии — более высокий. Например, после обслуживания запроса по линии REQ2 порядок приоритетов, начиная с наивысшего, становится таким: REQ3, REQ4, REQ1, REQ2. Ваша схема должна генерировать четыре выходных разрешающих сигнала, от GR1 до GR4, по одному для каждой входной линии запроса. В ответ на получение сигнала по линии DECIDE должен активизироваться один из этих сигналов.

4.14. Процессор 68000 имеет три линии IPL2-0, которые используются для запросов прерываний. На этих линиях 3-разрядное двоичное число интерпретируется процессором как представляющее устройство с наивысшим приоритетом, запросившее прерывание. Разработайте схему кодирования приоритетов, получающую запросы прерываний от семи устройств и генерирующую 3-разрядный код, представляющий запрос с наивысшим приоритетом.

4.15. (Данная задача подходит для лабораторного эксперимента.) Предположим, что в вашей лаборатории имеется подключенный к компьютеру видеотерминал.

а) Напишите программу ввода-вывода A, представляющую буквы в алфавитном порядке. Она выводит две следующие строки и затем останавливается:

ABC...YZ
ABC...YZ

б) Напишите программу ввода-вывода B, три раза подряд выводящую цифры от 0 до 9 в порядке возрастания. Ее выход должен иметь следующий формат:

012...9012...9012...9

Используйте программу A как главную программу, а программу B — как программу обработки прерывания, выполнение которой инициируется вводом с клавиатуры любого символа. Программа B тоже может прерываться вводом с клавиатуры какого-либо символа. Когда выполнение программы B завершается, возобновляется реализация последней прерванной программы (с точки прерывания). Программа B должна выполнять переход на новую строку таким образом, чтобы выходные данные выглядели следующим образом:

ABC
012...901
012...9012...9012...9
2...9012...9
DE...YZ

Для того чтобы начать новую строку, программе нужно вывести два символа: CR (OD_{16}) и LF(OA_{16}). Покажите, как можно использовать приоритет процессора для разрешения или запрета вложенных прерываний.

4.16. (Эта задача подходит для лабораторного эксперимента.) Когда в упражнении 4.15 вывод последовательности символов прерывается, то ее возобновление начинается с новой строки. Добавьте функцию перемещения курсора, чтобы при возобновлении вывода символы отображались в той же позиции, но в следующей строке. Таким образом, результат работы программы должен выглядеть так:

```
ABC
012...901
012...9012...9012...9
          2...9012...9
     DE...YZ
```

Модифицируйте программы, написанные для упражнения 4.15, таким образом, чтобы в случае прерывания вызывалась третья управляющая программа, С. Эта программа должна вызывать программу В для вывода последовательности символов. После этого и перед возвратом в прерванную программу она должна переместить курсор в нужную позицию.

4.17. В разделе 4.2.5 рассказывалось о точках останова. В тех местах, куда пользователь помещает точки останова, команды программы заменяются командами программного прерывания. Перед возвратом в исходную программу отладчик возвращает на место исходную команду программы, удаляя таким образом точку останова. Поясните, как отладчик может вернуть исходную команду программы на место, выполнить ее, а затем снова установить точку останова перед выполнением очередной команды программы.

4.18. Команда программного прерывания SWI процессора ARM может использоваться программой для вызова операционной системы и запроса определенного сервиса. Запрошенный сервис определяется восемью младшими разрядами команды. Каждый сервис операционной системы выполняется определенной подпрограммой, а начальные адреса этих подпрограмм хранятся в специальной таблице.

а) Приведите одну или более команд, которые могут использоваться операционной системой для копирования 8 младших разрядов команды SWI в регистр.

б) Приведите одну или более команд для вызова подпрограммы заданного сервиса.

4.19. Линия запроса прерывания, для которой используется схема с открытым коллектором, передает сигнал, представляющий собой логическое ИЛИ запросов от всех соединенных с ней устройств. В другой ситуации линия запроса прерывания должна генерировать сигнал, сообщающий о готовности

всех подключенных к шине устройств. Поясните, как для этого может использоваться схема с открытым коллектором.

4.20. В некоторых компьютерах процессор отвечает только на передний фронт сигнала запроса прерывания, поступающего по одной из его линий. Что произойдет, если к этой линии будут подключены два независимых устройства?

4.21. В приведенной на рис. 4.20 схеме устройство становится хозяином шины только в том случае, если уровень сигнала на входе, предоставляющего ему шину, изменяется от низкого к высокому. Предположим, что устройство 1 запрашивает шину и получает ее в свое распоряжение. Пока оно использует шину, устройство 3 активизирует свой выходной сигнал BR. Создайте временную диаграмму, показывающую, как устройство 3 становится хозяином шины, после того как ее освобождает устройство 1.

4.22. Обратимся к схеме арбитража шины, показанной на рис. 4.20. Предположим, что процессор удерживает сигнал BG1 активным, пока активен сигнал $\overline{\text{BR}}$. Когда устройство i запрашивает шину, оно становится ее хозяином только после перехода от низкого к высокому уровню сигнала на входе BGi.

а) Предположим, что устройства могут активизировать сигнал BR в любое время. Приведите последовательность событий, показывающую, что система может оказаться в состоянии взаимоблокировки, когда одно или несколько устройств запрашивают шину, шина свободна, но ни одно устройство не может стать ее хозяином.

б) Предложите правило поведения устройств, предотвращающее взаимоблокировки.

4.23. На рис. У4.1 приведена гирляндная схема, в которой сигнал запроса шины возвращается обратно в виде сигнала для ее предоставления. Предположим, что устройство 3 запрашивает шину и начинает ее использовать. Закончив работу, оно снимает сигнал BR3. Предположим, что время задержки при прохождении сигнала от узла BGi к узлу BG(i+1) любого устройства равно d. Покажите, что ложный сигнал предоставления шины будет возвращаться от устройства 3 (ложным он будет потому, что не является ответом ни на один запрос). Оцените ширину данного импульса.

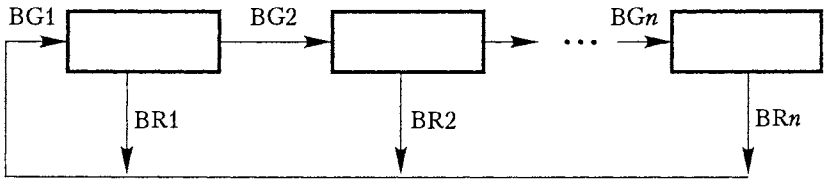


Рис. У4.1. Децентрализованная схема предоставления шины

4.24. После того как устройство 3 в упражнении 4.23 освободит шину, ее вскоре одновременно запросят устройства 1 и 5. Объясните, каким образом они оба получат управление шиной.

4.25 Обратимся к схеме арбитража, приведенной на рис. 4.20. Предположим, что локальный сигнал BUSREQ в интерфейсной схеме устройства равен 1, когда устройству требуется шина. Разработайте часть интерфейсной схемы, с входами BUSREQ, Bg*i*, $\overline{\text{BBSY}}$ и выходами $\overline{\text{BR}}$, BG(*i*+1), $\overline{\text{BBSY}}$.

4.26. Рассмотрим арбитражную схему, приведенную на рис. 4.22. Предположим, что код приоритета устройства хранится в интерфейсной схеме регистра. Разработайте схему для реализации такого арбитражного алгоритма. Арбитраж должен начинаться с активизации сигнала $\overline{\text{Start-Arbitration}}$. Чуть позже арбитражная схема, выигравшая цикл арбитража, должна активизировать выход, называемый Winner.

4.27. Как изменится показанная на рис. 4.26 временная диаграмма, если будет увеличено расстояние между процессором и устройством ввода-вывода? А как это увеличение отразится на схеме, представленной на рис. 4.24?

4.28. На промышленных предприятиях используются сенсоры для мониторинга температуры, давления и других факторов. Выход каждого сенсора состоит из ключа ON/OFF. Восемь таких сенсоров должны быть подключены к шине небольшого компьютера. Разработайте такой интерфейс, чтобы состояние всех восьми ключей можно было одновременно прочитать в один байт по адресу $FE10_{16}$. Предполагается, что шина работает в синхронном режиме и что для нее используется тактовая схема, показанная на рис. 4.24.

4.29. Разработайте интерфейс для подключения 7-сегментного индикатора к синхронной шине в качестве выходного устройства. (Схема 7-сегментного индикатора приведена на рис. A.37 в приложении A.)

4.30. Добавьте в интерфейс, приведенный на рис. 4.29, функцию прерываний. Покажите, как добавить бит разрешения прерываний, который может устанавливаться и очищаться процессором как разряд 6 регистра состояния интерфейса. Когда прерывания разрешены и входные данные доступны для чтения процессором, интерфейс должен активизировать линию запроса прерывания $\overline{\text{INTR}}$.

4.31. Для шины процессора используется схема с несколькими тактами, описанная в разделе 4.5.1. Быстродействие модуля памяти таково, что операция чтения может быть представлена временной диаграммой, приведенной на рис. 4.25. Разработайте интерфейсную схему для подключения такого модуля памяти к шине.

4.32. Давайте рассмотрим операцию записи данных через шину, описанную в разделе 4.5.1. Предположим, что процессор может передать адрес и данные на первом такте транзакции шины. Для сохранения данных в памяти потребуется два такта.

а) Может ли шина в течение этого времени использоваться для других транзакций?

б) Можно ли в этом случае обойтись без ответных сигналов из памяти? (Подсказка: внимательно проанализируйте ситуацию, когда процессор пытается выполнить другую операцию записи в этот же модуль памяти, в то время как этот модуль занят выполнением предыдущего запроса. Объясните, как нужно поступать в подобной ситуации.)

4.33. На рис. 4.24–4.26 продемонстрированы три разных подхода к разработке шины. Что произойдет в каждом из этих случаев, если адресуемое устройство не ответит из-за сбоев в функционировании? Какие это вызовет проблемы и как их можно будет разрешить?

4.34. В случае временной диаграммы, приведенной на рис. 4.25, процессор сохраняет на шине адрес до тех пор, пока не получит от устройства ответ. Необходимо ли это? Какие дополнения потребуются со стороны устройства, если процессор будет сохранять адрес активным лишь в течение одного такта?

4.35 Рассмотрим синхронную шину, работающую в соответствии с временной диаграммой, приведенной на рис. 4.24. Переданный процессором адрес появляется на шине через 4 нс. Время задержки на распространение сигнала по проводам шины между процессором и различными устройствами варьируется от 1 до 5 нс, декодирование адреса занимает 6 нс, а еще от 5 до 10 нс у адресного устройства уходит на помещение на шину запрошенных данных. На установку входного буфера требуется 3 нс. Какова максимальная тактовая частота, на которой может работать эта шина?

4.36. Время, необходимое для полной пересылки данных по шине, которая представлена на рис. 4.26, варьируется в зависимости от задержек. Рассмотрим шину с теми же параметрами, что указаны в упражнении 4.35. Какова максимальная и минимальная длительность цикла шины?

Глава 5
Система памяти

- ✦ Базовые схемы памяти
- ✦ Организация основной памяти
- ✦ Концепция кэш-памяти
- ✦ Виртуальная память
- ✦ Магнитные и оптические диски, магнитные ленты

Программы и обрабатываемые ими данные хранятся, как известно, в памяти компьютера. В этой главе мы поговорим о том, как функционирует эта жизненно важная часть компьютерной системы. Читатель уже знает, что скорость выполнения программ напрямую зависит от скорости передачи данных между процессором и памятью и что для выполнения больших программ, обрабатывающих огромные массивы данных, необходима память очень большого объема.

В идеале память должна быть быстрой, большой и недорогой. Однако удовлетворить всем трем требованиям одновременно, к сожалению, невозможно. Чем больше память и чем быстрее она работает, тем дороже она стоит. Поэтому проектировщики компьютерных систем трудятся над разработкой и усовершенствованием технологий, позволяющих создавать для компьютера видимость большой и быстрой памяти.

Мы начнем эту главу с рассказа о наиболее распространенных компонентах и структурах, используемых для реализации памяти, а затем поговорим о быстродействии памяти и о том, как ее можно увеличить за счет кэширования. Далее речь пойдет о концепции виртуальной памяти, позволяющей представить память, как ее видит процессор, то есть большей, чем на самом деле. А напоследок мы расскажем вам о вторичных запоминающих устройствах, емкость которых значительно больше емкости основной памяти.

5.1. Базовые концепции

Максимальный размер памяти, который может использоваться компьютером, определяется его системой адресации. К примеру, 16-разрядный компьютер, генерирующий 16-разрядные адреса, может иметь память объемом до $2^{16} = 64$ Кбайт адресуемых единиц хранения, компьютер, команды которого генерируют 32-разрядные адреса, может использовать память объемом до $2^{32} = 4$ Гбайт адресуемых единиц, а для компьютеров с 40-разрядными адресами доступная память объемом

до 2^{32} = 1 Тбайт адресуемых единиц. Количество адресуемых единиц памяти определяет размер ее адресного пространства.

Большинство современных компьютеров, как вы знаете, адресуют память побайтово. На рис. 2.7 показано возможное назначение адресов в 32-разрядном компьютере с байтовой адресацией. При этом в процессорах Intel используется прямой порядок байтов, а в процессоре 68000 — обратный. Архитектура процессора ARM позволяет использовать обе схемы. С точки зрения структуры памяти между этими схемами нет существенных различий.

Обычно память разрабатывается с учетом того, что данные извлекаются и считываются не байтами, а словами. Само понятие длины слова чаще всего определяется как количество битов, сохраняемых или считываемых за одно обращение к памяти. Возьмем, к примеру, компьютер с байтовой адресацией, команды которого генерируют 32-разрядные адреса. Когда процессор генерирует 32-разрядный адрес основной памяти, старшие 30 разрядов определяют слово, к которому производится доступ. Если задано и количество байтов, два младших разряда определяют его положение в слове. В байтовой операции чтения из памяти могут быть извлечены и другие байты, но они игнорируются процессором. Если же выполняется байтовая операция записи, управляющие схемы памяти должны гарантировать, что остальные байты слова останутся неизменными.

Современные схемы реализации компьютерной памяти довольно сложны. Чтобы упростить для вас процесс знакомства со структурами памяти, мы сначала обсудим традиционную архитектуру, и только после этого перейдем к современным подходам и технологиям.

Со стороны системы запоминающее устройство можно рассматривать как черный ящик. Пересылка данных между памятью и процессором выполняется, как вы помните из раздела 1.3, с помощью двух регистров процессора, обычно называемых MAR (Memory Address Register — регистр адреса) и MDR (Memory Data Register — регистр данных). Если регистр MAR содержит k битов, а регистр MDR — n битов, память может содержать до 2^k адресуемых единиц хранения. За один цикл обращения к памяти между нею и процессором пересылается n бит данных. Данные передаются по шине процессора, имеющей k адресных линий и n линий данных. Кроме того, шина содержит линии для управления передачей данных R/$\overline{\text{W}}$ (Read/$\overline{\text{W}}$rite) и MFC (Memory Function Compelled). Могут использоваться и другие линии, с помощью которых задается количество пересылаемых данных. Соединение между процессором и памятью схематически показано на рис. 5.1.

Чтобы считать данные из памяти, процессор сначала загружает адрес в регистр MAR и устанавливает линию R/$\overline{\text{W}}$ в 1. В ответ память помещает данные на линии данных и подтверждает это действие активизацией сигнала MFC. После получения сигнала MFC процессор загружает данные с адресных линий в регистр MDR.

Для того чтобы записать данные в память, процессор загружает адрес в регистр MAR, а данные в регистр MDR и устанавливает линию R/$\overline{\text{W}}$ в 0, указывая таким образом, что выполняется операция записи.

Если в операциях чтения производится обращение по последовательным адресам, может быть выполнена операция блочной пересылки, при которой памяти передается только один адрес — адрес первого байта блока данных. О таких операциях подробнее рассказывается в разделе 5.5.

Доступ к памяти может синхронизироваться тактовым генератором или описанными в разделе 4.5.1 специальными сигналами, управляющими пересылкой по шине. Управление операциями чтения из памяти и записи в память осуществляется так же, как и управление операциями ввода и вывода по шине.

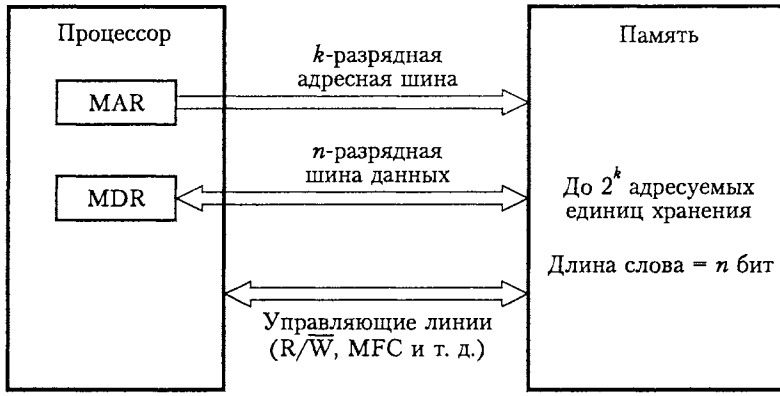


Рис. 5.1. Организация связи системы памяти с процессором

Быстродействие памяти характеризуется интервалом времени между инициированием операции и ее завершением, например временем между сигналами чтения и MFC. Это время определяют как *время доступа к памяти*. Еще одной важной характеристикой быстродействия памяти является *цикл памяти* — минимальный промежуток времени между моментами начала двух последовательных операций с памятью, например, между двумя последовательными операциями чтения. Цикл памяти обычно немного больше времени доступа (это зависит от особенностей реализации запоминающего устройства).

В запоминающем устройстве, называемом *памятью с произвольным доступом* (Random Access Memory, RAM), на обращение по любому адресу уходит одно и то же время. Этим RAM-память отличается от запоминающих устройств с последовательным или частично-последовательным доступом, таких как магнитные диски или ленты. Время доступа последних зависит от адреса или местоположения данных.

Для реализации основной памяти компьютера используются полупроводниковые интегральные схемы. В следующих разделах рассказывается о структуре и принципах работы такой памяти, а также о технологиях, позволяющих увеличить ее быстродействие и объем.

Обычно процессор обрабатывает команды и данные быстрее, чем они выбираются из памяти. Поэтому цикл доступа к памяти является узким местом системы. Одним из способов сокращения времени доступа к памяти может стать использование *кэш-памяти*. Это быстрая память небольшого объема, расположенная между сравнительно медленной основной памятью компьютера и процессором. В ней хранятся текущие фрагменты программы и ее данных.

Еще одной важной концепцией, связанной с организацией памяти, является концепция *виртуальной памяти*. До сих пор мы предполагали, что генерируемые

процессором адреса полностью соответствуют физическим ячейкам памяти. Однако на практике это не всегда так. По определенным причинам, о которых мы поговорим далее в этой главе, данные могут храниться не по тем адресам, которые заданы в программе. Схемы управления памятью преобразуют указанный в программе адрес в другой адрес, используемый для доступа к физической памяти. В таком случае сгенерированный процессором адрес называют *виртуальным* или *логическим*. Виртуальное адресное пространство определенным образом отображается на физическую память, в которой хранятся данные. Отображение выполняется с помощью специальных управляющих схем памяти, часто называемых *блоком управления памятью*. В ходе выполнения программы функция отображения может меняться в соответствии с системными требованиями.

Виртуальная память предназначена для увеличения видимого компьютером объема физической памяти. Виртуальное адресное пространство и объем расположенных в нем данных могут быть настолько большими, насколько позволяют возможности адресации используемого процессора. Причем в каждый конкретный момент только часть этого адресного пространства отображается на физическую память. Остальные виртуальные адреса соответствуют устройствам массовой памяти — как правило, магнитным дискам. Когда выполняющейся программе требуется доступ к данным, адреса которых не отображаются на реальную память, блок управления памятью изменяет функцию отображения и пересылает данные с диска в основную память. Поэтому в каждом цикле обращения к памяти система обработки адресов определяет, находится ли адресуемая информация в основной памяти компьютера. Если да, происходит обращение к соответствующему слову памяти и выполнение программы продолжается. Если нет, с диска в память пересылается *страница*, содержащая нужное слово (см. раздел 5.7.4). Эта страница заменяет в памяти какую-нибудь другую страницу, которая пока не нужна программе. Поскольку на пересылку страниц между памятью и диском уходит некоторое время, при частом перемещении страниц скорость работы компьютера снижается. Но если выбор заменяемых страниц выполняется продуманно, вероятность того, что необходимые страницы окажутся в основной памяти, возрастет.

В этом разделе будет коротко рассказано о нескольких важнейших функциях организации системы памяти. Их назначение заключается в обеспечении компьютерной системы настолько большой и быстрой памятью, насколько это возможно с учетом общей стоимости системы. Мы не ждем, что читатель сразу же усвоит все концепции организации памяти и их следствия, поэтому к этим вопросам мы еще вернемся в данной главе. Пока же мы лишь хотим ввести основные понятия укаханной области в их взаимосвязи, поскольку это так же важно, как изучение каждой отдельной функции и концепции.

5.2. Полупроводниковая RAM-память

Полупроводниковая память реализуется в виде микросхем с очень разным быстродействием. Длительность их цикла варьируется от 100 до менее чем 10 нс. Появившиеся в конце 1960-х полупроводниковые схемы памяти были гораздо дороже памяти на магнитных сердечниках. Однако стремительное развитие технологии

СБИС (сверхбольших интегральных схем) позволило быстро снизить их стоимость, и теперь практически вся память реализуется в виде полупроводниковых микросхем. В этом разделе рассказывается о важнейших характеристиках полупроводниковой памяти. Для начала мы поговорим о том, как объединяется в одной микросхеме множество ячеек памяти.

5.2.1. Организация микросхем памяти

В каждой ячейке памяти, которые обычно объединяются в массивы, может храниться один бит информации. На рис. 5.2 показано, как может быть организован такой массив. Каждая строка ячеек составляет слово памяти, а все ячейки строки соединяются общей линией, называемой *линией слова*, которая управляется входящим в состав того же чипа дешифратором адреса. Ячейки каждого столбца соединяются со схемой Sence/Write двумя *линиями битов*. Схемы Sence/Write соединяются линиями записи и считывания данных. Во время операции чтения схемы считывают информацию из ячеек, выбранных с помощью линии слова, и пересылают ее на выходные линии данных. А в процессе операции записи они получают входную информацию и сохраняют ее в ячейках выбранного слова.

На рис. 5.2 показана организация очень маленькой микросхемы памяти, состоящей всего из 16 слов по 8 бит в каждом. Структуру такой микросхемы обозначают как 16 × 8. Входы и выходы данных каждой схемы Sence/Write соединяются с одной двунаправленной линией данных, которая может быть подключена к шине компьютера. В дополнение к линиям адреса и данных имеются две управляющие линии, R/$\overline{W}$ и CS. Входное значение на линии R/$\overline{W}$ (Read/$\overline{W}$rite) определяет требуемую операцию, а входное значение на линии CS (Chip Select — выбор элемента памяти) выбирает одну из микросхем, составляющих систему памяти.

Схема памяти, приведенная на рис. 5.2, вмещает 128 бит данных и требует 14 внешних линий для адреса, данных и управляющих сигналов. Кроме того, для нее, конечно же, потребуются еще две линии, которые должны соединить ее с источником питания и «землей». Рассмотрим чуть большую микросхему памяти, содержащую 1 К (1024) ячеек памяти. Такая схема может быть организована в виде массива 128 × 8, и для нее потребуется 19 внешних соединений. В качестве альтернативы то же количество ячеек можно организовать в массив 1 К × 1. Тогда понадобятся 10-разрядный адрес, но зато лишь одна линия данных, а следовательно, 15 внешних соединений. Эта организация показана на рис. 5.3. Здесь 10-разрядный адрес делится на две группы по пять разрядов в каждой, представляющих адреса строки и столбца в массиве ячеек. Сигнал адреса строки задает строку из числа 32 параллельно доступных ячеек. Однако в соответствии с адресом столбца только одна из них соединяется с внешней линией данных с помощью выходного мультиплексора и входного демультиплексора.

Современные микросхемы памяти содержат гораздо большее количество ячеек, чем представленные на рис. 5.2 и 5.3. Такие маленькие схемы мы использовали просто для наглядности. Большие микросхемы имеют ту же структуру, но содержат массивы большего размера и имеют большее число внешних соединений. Например, 4-мегабайтовая микросхема может иметь структуру 512 К × 8, для которой понадобятся 19 адресных выводов и 8 выводов для ввода-вывода данных. В настоящее время выпускаются микросхемы емкостью в сотни мегабитов.

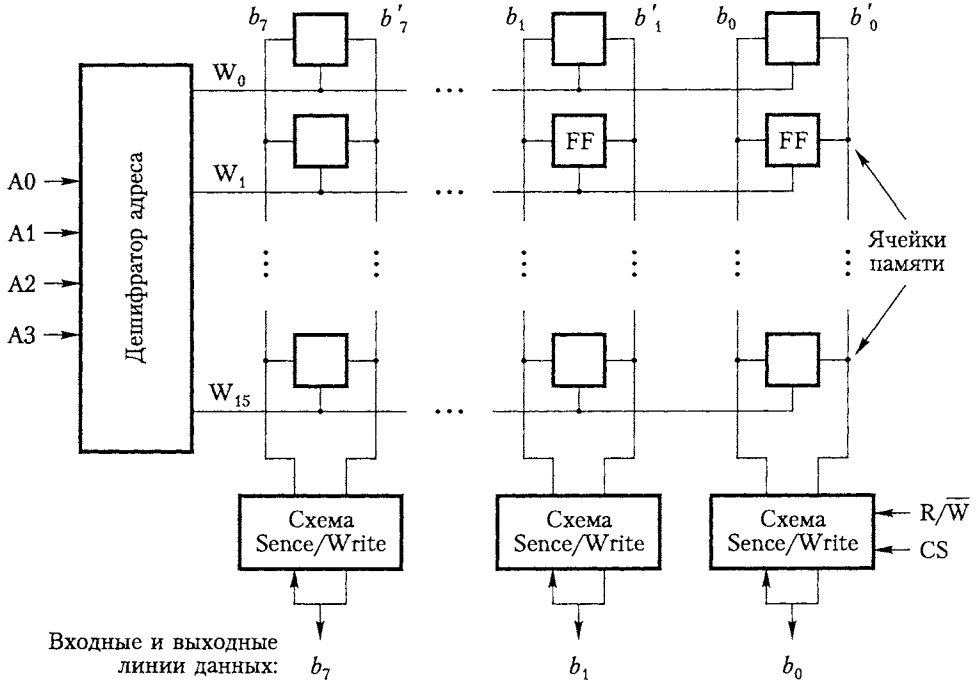


Рис. 5.2. Организация битовых ячеек в микросхеме памяти

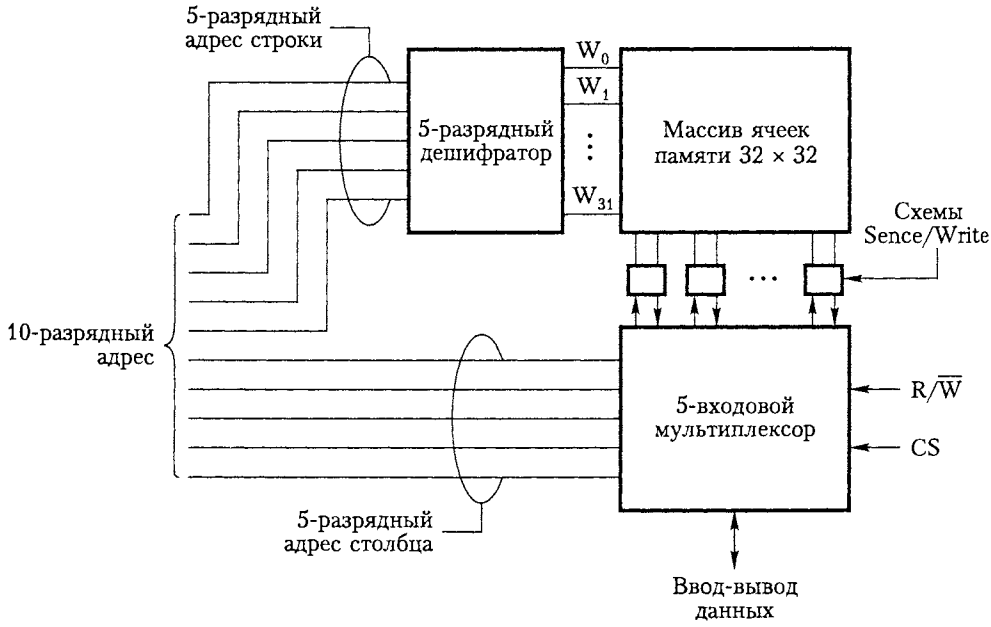


Рис. 5.3. Микросхема памяти 1 K × 1

5.2.2. Статическая память

Память на основе микросхем, которые могут сохранять свое состояние лишь до тех пор, пока к ним подключено питание, называется *статической* (Static RAM, SRAM). Как может быть реализована такая память, показано на рис. 5.4. Перекрестным соединением двух инверторов образуется защелка. Эта защелка соединяется с двумя линиями битов посредством транзисторов T_1 и T_2. Транзисторы действуют как переключатели, которые могут открываться и закрываться под управлением линии слова. Когда на линии слова задана «земля», транзисторы выключены и состояние защелки не меняется. Для примера предположим, что ячейка находится в состоянии 1, если логическое значение в точке X равно 1, а в точке $Y - 0$. Это состояние сохраняется до тех пор, пока сигнал на линии слова находится на уровне «земли».

Операция чтения

Для того чтобы прочитать состояние ячейки SRAM, схемы управления памятью активизируют линию слова, в результате чего закрываются ключи T_1 и T_2. Если значение в ячейке равно 1, на линии b наблюдается высокий уровень сигнала, а на линии b' — низкий. Если же значение в ячейке равно 0, эти сигналы меняют свои значение на противоположное. Схемы Sence/Write на концах линий битов выполняют мониторинг состояния линий b и b' и соответствующим образом устанавливают выходные сигналы.

Операция записи

Для установки состояния ячейки соответствующее значение помещается на линию b, его дополнение — на линию b', а затем активизируется линия слова. Необходимые сигналы на линиях битов генерируются схемой Sence/Write.

Ячейка КМОП

КМОП-реализация ячейки, показанной на рис. 5.4, приведена на рис. 5.5. Пары транзисторов T_3, T_5 и T_4, T_6 образуют инверторы защелки (см. приложение А). Состояние ячейки считывается или записывается так, как описано выше. Например, в состоянии 1 напряжение в точке X сохраняется высоким за счет того, что транзисторы T_3 и T_6 включены, а транзисторы T_5 и T_4 выключены. Таким образом, если транзисторы T_1 и T_2 включены (замкнуты), напряжение на линиях битов b и b' будет соответственно высоким и низким.

В старых КМОП-микросхемах статической памяти напряжение источника питания V_{supply} составляло 5 В, а в новых низковольтных микросхемах оно равно 3,3 В. Обратите внимание, что для сохранения состояния ячейки необходимо обеспечить постоянное питание. Если питание отключить, содержимое ячейки будет уничтожено. Когда питание будет подано снова, защелка установится в устойчивое состояние, но это не обязательно будет то самое состояние, в каком она была в момент отключения питания. Поэтому микросхемы статической памяти называют *энергозависимыми*.

Основным преимуществом статической КМОП-памяти является очень низкая потребляемая мощность. Через ячейки этой памяти ток идет только в момент

обращения к ним. Все остальное время транзисторы T_1 и T_2, а также по одному транзистору в каждом инверторе выключены, и между источником питания V_{supply} и «землей» нет соединения.

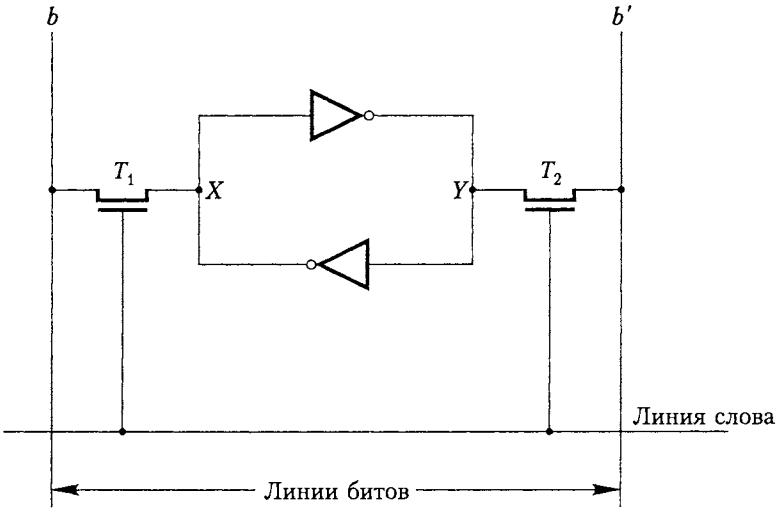


Рис. 5.4. Ячейка статической RAM

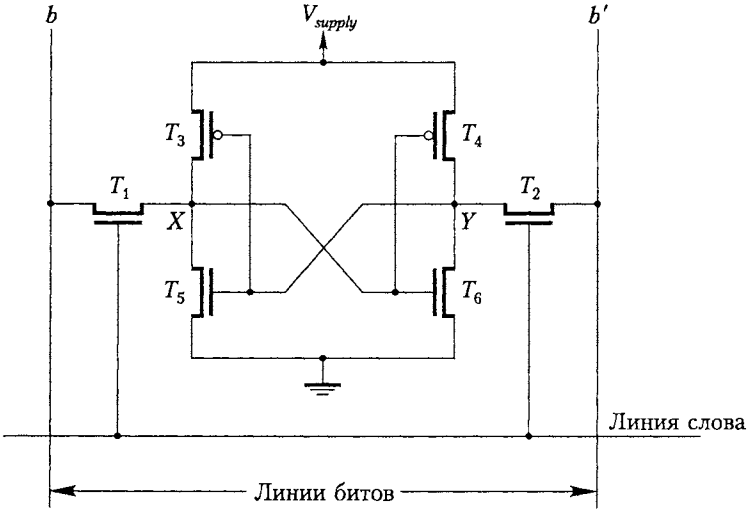


Рис. 5.5. Пример ячейки памяти КМОП

В современных микросхемах время доступа к статической памяти составляет всего несколько наносекунд. Поэтому статическая память используется в первую очередь там, где особенно важен такой показатель, как высокая скорость работы.

5.2.3. Асинхронная динамическая память

Статическая RAM работает быстро, но стоит очень дорого, поскольку каждая ее ячейка содержит несколько транзисторов. Вот почему выпускается еще и более дешевая память с более простой конструкцией ячеек. Однако эти ячейки не способны бесконечно долго сохранять свое состояние, поэтому такая память называется *динамической* (Dynamic RAM, DRAM).

В ячейке динамической памяти информация хранится в форме заряда на конденсаторе, и этот заряд может сохраняться всего несколько десятков миллисекунд. Поскольку ячейка памяти должна хранить информацию гораздо дольше, ее содержимое должно периодически обновляться путем восстановления заряда на конденсаторе.

На рис. 5.6 показан пример ячейки динамической памяти, состоящей из конденсатора *C* и транзистора *T*. Для записи информации в эту ячейку включается транзистор *T* и на линию бита подается соответствующее напряжение. В результате на конденсаторе образуется определенный заряд.

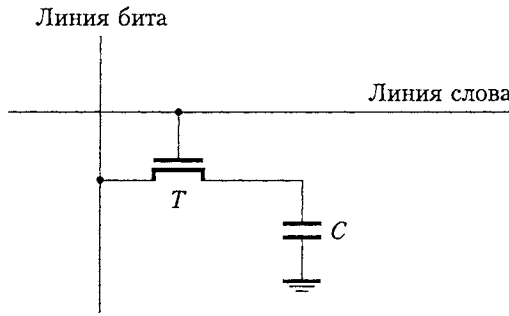


Рис. 5.6. Однотранзисторная ячейка динамической памяти

После выключения транзистора конденсатор начинает разряжаться. Это происходит из-за его собственного сопротивления утечки, а также из-за того, что после выключения транзистор продолжает слабо проводить ток (измеряется в пикоамперах). Полученная информация не содержит ошибок лишь в том случае, если она считывается из ячеек до того, как заряд конденсатора падает ниже определенного порогового значения. Операция чтения производится, когда транзистор выбранной ячейки включен. Соединенный с линией бита усилитель считывания определяет, превышает ли заряд конденсатора пороговое значение. Если да, он подает на линию бита напряжение, соответствующее значению 1. В результате конденсатор заряжается до напряжения, также соответствующего 1. Если заряд на конденсаторе ниже порогового значения, усилитель считывания снижает напряжение на линии бита до уровня «земли», обеспечивая тем самым отсутствие заряда (логическое значение 0) на конденсаторе. Таким образом, в процессе считывания содержимое ячейки автоматически обновляется. Все ячейки выбранной строки считываются одновременно, в результате чего обновляется содержимое всей строки. Подробности реализации усилителя считывания в этой книге не рассматриваются.

На рис. 5.7 показана 16-мегабитная микросхема DRAM конфигурации 2M × 8. Ее ячейки организованы в массив 4 К × 4 К, в котором 4096 ячеек каждой строки разделены на 512 групп по 8 ячеек, так что в одной строке может храниться 512 байт данных. Следовательно, для выбора строки требуется 12 адресных разрядов. Еще 9 разрядов необходимо для выбора в строке группы из 8 бит. Значит, для доступа к байту в такой микросхеме нужен 21-разрядный адрес. Старшие 12 и младшие 9 разрядов адреса составляют адреса строки и столбца байта. Для сокращения количества выводов микросхемы адреса строки и столбца мультиплексируются на 12 выводов. В процессе операции чтения или записи сначала на адресные выводы микросхемы подается адрес строки. В ответ на входной сигнал RAS (Row Address Strobe — строб адреса строки) он загружается в защелку адреса строки. Затем инициируется операция чтения, в ходе которой считываются и обновляются ячейки выбранной строки. Через некоторое время после загрузки адреса строки на адресные выводы подается адрес столбца, который загружается в защелку адреса столбца в ответ на сигнал CAS (Column Address Strobe — строб адреса столбца). Информация из этой защелки декодируется и выбирается соответствующая группа из 8 схем Sense/Write. Если управляющий сигнал R/$\overline{W}$ указывает на операцию считывания, выходные значения выбранных схем пересылаются на линии данных, D_{7-0}. Для операции записи информация с линий D_{7-0} пересылается в схемы. Затем она используется для перезаписи содержимого указанных ячеек в соответствующих 8 столбцах. В коммерческих микросхемах активизации сигналов RAS и CAS соответствует низкий уровень напряжения, так что стробирование адреса выполняется при переходе соответствующего сигнала от высокого уровня к низкому. На схемах эти сигналы обозначаются как $\overline{RAS}$ и $\overline{CAS}$.

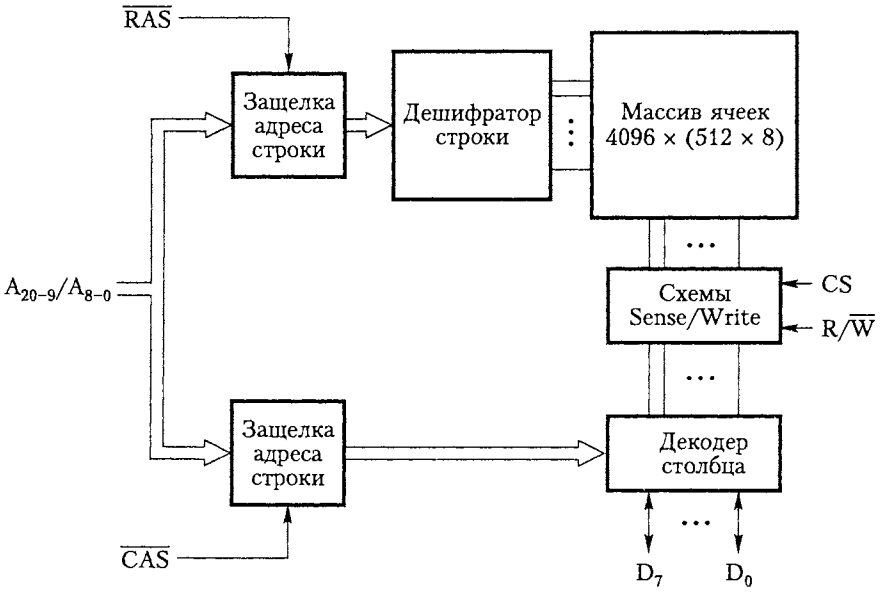


Рис. 5.7. Внутренняя организация микросхемы динамической памяти 2M × 8

Подача адреса строки в ходе операции считывания или записи приводит к чтению и обновлению всех ячеек этой строки. Для того чтобы поддерживать содержимое памяти DRAM, нужно периодически обращаться к каждой ее строке. Обычно эта работа автоматически выполняется с помощью специальной схемы, называемой *схемой регенерации*. Схемы регенерации часто интегрируют прямо в микросхемы динамической памяти, чтобы их динамическая природа была практически невидимой для пользователя.

Описанная в этом разделе динамическая память управляется в асинхронном режиме. Она тактируется управляющими сигналами RAS и CAS, которые генерируются специальной схемой управления памятью. При этом процессор должен учитывать задержку ответа памяти. Такая память называется *асинхронной DRAM*. Благодаря своей высокой емкости (от 1 до 256 Мбит) и дешевизне микросхемы DRAM широко используются в запоминающих устройствах компьютеров. Ведется работа по созданию и более емких микросхем. Для сокращения количества микросхем в компьютере DRAM организуется таким образом, чтобы при выполнении операции чтения или записи биты пересылались параллельно (рис. 5.7). Микросхемы имеют разную организацию, благодаря чему из них можно свободно компоновать любые системы памяти. Например, микросхема объемом 64 Мбит может быть организована как 16 М × 4, 8 М × 8 или 4 М × 16.

Быстрый постраничный режим

При обращении к микросхеме DRAM, показанной на рис. 5.7, считывается содержимое всех 4096 ячеек выбранной строки, но на линии данных D_{7-0} помещаются только 8 бит. Этот байт выбирается битами A_{8-0} адреса столбца. Данную схему можно немного модифицировать, что позволит обращаться к другим байтам той же строки, не выбирая ее повторно. С этой целью на выход усилителя считывания каждого столбца нужно добавить по защелке. При выборе нового адреса строки будут устанавливаться защелки для всех ее битов. Теперь для помещения нужного байта на выходные линии данных достаточно установить соответствующий адрес столбца.

Обычно байты пересылаются последовательно, в порядке возрастания их адресов. Для того чтобы как можно быстрее переслать блок таких данных, нужно под управлением ряда сигналов CAS прямо в схеме генерировать последовательные номера столбцов. Такой режим блочной пересылки называется *быстрым постраничным режимом* (Fast Page Mode, FPM). Согласно популярной сленговой терминологии, небольшие группы байтов называются блоками, а большие — страницами.

Ускоренная блочная пересылка данных особенно полезна в тех системах, где данные большими массивами пересылаются в память и из памяти, скажем, в графических терминалах. В компьютерах общего назначения она применяется для пересылки данных между основной памятью и кэшем (см. раздел 5.5).

5.2.4. Синхронная DRAM

Результатом последних разработок в области технологий памяти стало создание DRAM, синхронизируемой тактовым сигналом. Она получила название *синхронная DRAM* (Synchronous DRAM, SDRAM), а ее структура показана на рис. 5.8.

Как видите, массив ячеек в ней точно такой же, как в асинхронной DRAM. Линии адреса и данных буферизируются посредством регистров. Выход каждого усилителя считывания соединен с защелкой — обратите на это особое внимание. В ходе операции чтения в эти защелки загружается содержимое всех ячеек выбранной строки. Однако если обращение к строке выполняется только с целью регенерации данных, содержимое защелок не меняется — производится только регенерация ячеек. Данные из защелок, соответствующих выбранным столбцам, пересылаются в выходные регистры данных, откуда они могут быть прочитаны через выводы, предназначенные для выходных данных.

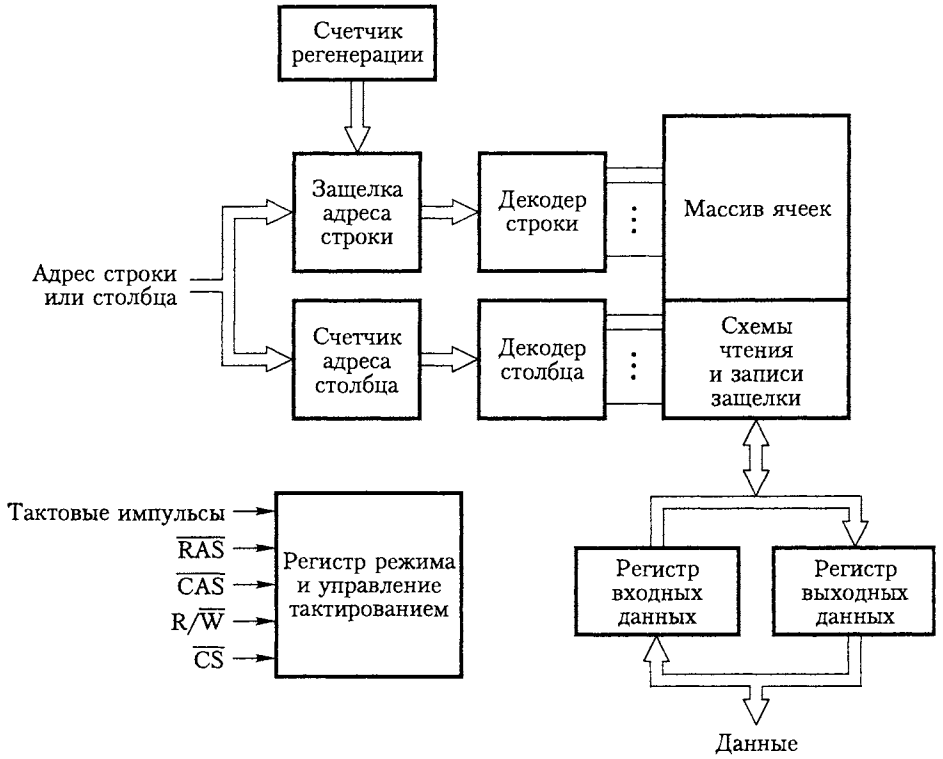


Рис. 5.8. Синхронная DRAM

Память SDRAM может функционировать в нескольких режимах, определяемых управляющей информацией в регистре *режима*. Например, могут задаваться пакетные операции для передачи различных объемов данных. В пакетных операциях применяется описанный выше режим блочной пересылки данных FPM. В SDRAM для выбора последовательных столбцов не обязательно использовать внешние импульсы на линии CAS. Управляющие сигналы можно генерировать прямо внутри схемы — на основе значений счетчика столбцов и тактового сигнала. В этом случае новые данные помещаются на линии данных на каждом такте. Все действия выполняются на переднем фронте тактового сигнала.

На рис. 5.9 приведена временная диаграмма типичной операции пакетного чтения длительностью 4 такта. Первым делом в ходе этой операции по сигналу $\overline{RAS}$ фиксируется адрес строки. На активизацию выбранной строки уходит два или три такта (на рисунке показано два такта). Затем по сигналу $\overline{CAS}$ фиксируется адрес столбца. После задержки в один такт на линию данных помещается первый набор битов данных. Для обращения к следующим трем наборам битов выбранной строки, помещаемым на линии данных на следующих трех тактах, SDRAM автоматически увеличивает адрес столбца.

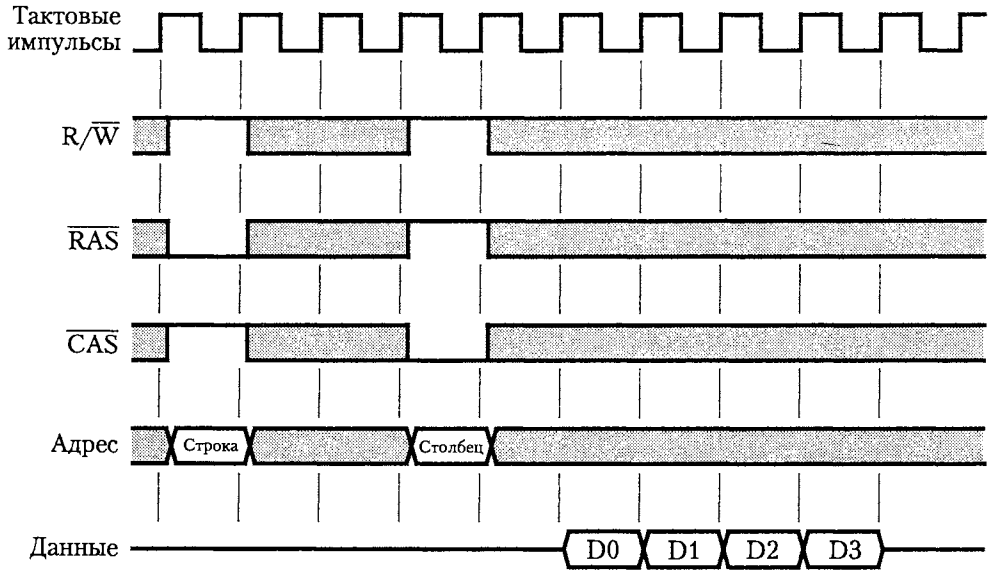


Рис. 5.9. Временная диаграмма операции пакетного чтения из SDRAM

В SDRAM имеется встроенная схема регенерации. В состав этой схемы входит счетчик, формирующий адрес строки, которая выбрана для регенерации. В типичной SDRAM данные регенерируются по меньшей мере каждые 64 мс.

Современные SDRAM могут работать на тактовой частоте до 100 МГц и, как правило, предназначены для серийных процессоров. Например, Intel определила спецификации шин PC100 и PC133, согласно которым системная шина (к ней подключена основная память) управляется тактовым сигналом с частотой 100 и 133 МГц. Поэтому ведущие производители микросхем памяти выпускают чипы SDRAM с частотой 100 и 133 МГц.

Время ожидания

Данные между памятью и процессором, а точнее, между памятью и кэшем процессора, пересылаются в виде слов или небольших блоков слов. Большие блоки, составляющие страницы данных, пересылаются между памятью и дисками — об этом речь пойдет в разделе 5.7. Скорость и эффективность таких операций заметно отражаются на производительности всей компьютерной системы. Производительность

удобно характеризовать двумя параметрами: временем ожидания и пропускной способностью.

Термином *время ожидания памяти* или *латентность* (latency) определяется время, уходящее на пересылку в память или из памяти одного слова данных. Если данные считываются и записываются по одному слову, латентность полностью характеризует производительность памяти. Однако для пакетных операций, в ходе которых пересылаются блоки данных, полное время, уходящее на выполнение операции, зависит от скорости пересылки отдельных слов и размера блока данных. Поэтому при блочной пересылке под временем ожидания подразумевается время пересылки первого слова данных. Обычно это слово пересылается значительно дольше следующих слов блока. Например, на временной диаграмме, представленной на рис. 5.9, цикл доступа к памяти начинается с активизации сигнала $\overline{\text{RAS}}$. По прошествии пяти тактов пересылается первое слово данных. Таким образом, время ожидания составляет пять тактов. При тактовой частоте 100 МГц это 50 нс. Оставшиеся три слова пересылаются на последовательных тактах.

Конечно же, нам важно, сколько времени уходит на пересылку всего блока данных. Поскольку блоки имеют разную длину, производительность можно определять количеством битов или байтов, пересылаемых за одну секунду. Эту характеристику называют *пропускной способностью* (bandwidth) памяти. Пропускная способность блока памяти, состоящего из одной или более микросхем, зависит от скорости доступа к хранящимся в памяти данным и от количества параллельно доступных битов. Однако реальная пропускная способность компьютерной системы (с учетом пересылки данных между памятью и процессором) определяется не только быстродействием памяти; она зависит и от пропускной способности соединений между памятью и процессором, то есть (в типичном случае) от пропускной способности шины. Микросхемы памяти обычно разрабатываются с учетом скорости функционирования популярных шин. Очевидно, что пропускная способность зависит от скорости доступа и пересылки по одной линии, а также от количества параллельно пересылаемых битов, то есть от количества линий шины. Иными словами, пропускная способность является произведением скорости пересылки данных (и доступа к ним) и ширины шины данных.

DDR SDRAM

Рынок требует постоянного повышения производительности компьютерных систем, заставляя их разработчиков создавать все более быстрые версии микросхем памяти. Стандартная SDRAM выполняет все операции на переднем фронте тактового сигнала. Вслед за ней появилась память, доступ к ячейкам которой выполняется тем же способом, но данные пересылаются на обоих фронтах сигнала. Время ожидания таких микросхем то же, что и у стандартных SDRAM, но в случае больших пакетных операций пропускная способность почти вдвое выше. Такая память называется *SDRAM с удвоенной пропускной способностью* (Double Data Rate SDRAM, DDR SDRAM).

Для ускорения доступа к данным массив ячеек разделен на два независимых массива. Последовательные слова блока данных хранятся в разных массивах. Такое *чередование* слов позволяет одновременно считывать из памяти два слова, одно из которых пересылается на переднем, а другое — на заднем фронте тактового сигнала. В разделе 5.6.1 мы рассмотрим концепцию чередования подробнее.

DDR SDRAM и стандартные SDRAM наиболее эффективны в системах, где данные пересылаются преимущественно блоками. К системам такой категории относятся и компьютеры общего назначения, в которых пересылка данных выполняется между основной памятью и кэшем (раздел 5.5). Кроме того, блочная пересылка применяется в высококачественных видеодисплеях.

5.2.5. Структура памяти большого объема

Итак, мы обсудили базовую организацию схем памяти, которые можно реализовать на одной микросхеме. Теперь можно поговорить о том, как такие микросхемы соединяются в более крупные запоминающие устройства.

Системы статической памяти

Рассмотрим память, состоящую из 2 М (2097152) слов по 32 бита каждое. На рис. 5.10 показано, как реализовать такую память на основе микросхем статической памяти 512 К × 8. Каждый столбец на этом рисунке состоит из четырех микросхем, содержащих восемь последовательных битов каждого слова. Четыре таких набора составляют память 2 М × 32. У каждой микросхемы имеется управляющий вход, называемый CS (Chip Select — выбор микросхемы). Когда на этот вход подается 1, микросхема может принимать данные или помещать их на свои линии данных. Выход данных любой микросхемы имеет три состояния (см. раздел А.5.4). В каждый конкретный момент только одна микросхема помещает данные на выходные линии данных, а выходы всех остальных микросхем находятся в высокоимпедансном состоянии. Для выбора 32-разрядного слова из такой памяти необходим 21 адресный разряд. Два старших разряда определяют, какой из четырех управляющих сигналов CS следует активизировать, а оставшиеся 19 разрядов применяются для доступа к конкретному байту заданной строки внутри каждой микросхемы. Входы $R/\overline{W}$ всех микросхем соединяются вместе, образуя единый управляющий вход, не показанный на этом рисунке.

Системы динамической памяти

Большие системы динамической памяти имеют ту же структуру, что и память, представленная на рис. 5.10. Однако физически они чаще выполняются в виде более удобных *модулей памяти*.

Современным компьютерам необходима очень большая память. Даже маленький персональный компьютер, как правило, имеет хотя бы 32 Мбайт памяти, а типичная рабочая станция — как минимум 128 Мбайт. Чем больше основная память компьютера, тем выше его производительность, поскольку в памяти может храниться большее количество программ и обрабатываемых ими данных, а значит, меньше придется обращаться к внешней памяти. Но если все необходимые микросхемы DRAM будут размещены прямо на основной системной печатной плате, где содержится процессор (ее часто называют *материнской платой*), они займут слишком много места. Кроме того, будет затруднено дальнейшее наращивание памяти, поскольку для добавляемой памяти придется выделить дополнительное место на плате, а для установки микросхем нужно будет подвести соединения ко всем разъемам. Поэтому были разработаны модули памяти большего объема, называемые SIMM (Single In-Line Memory Module — модуль памяти с однорядным

расположением выводов) и DIMM (Dual In-Line Memory Module — модуль памяти с двухрядным расположением выводов). Такие модули представляют собой маленькие платы с наборами микросхем памяти, вертикально устанавливаемые в специальные разъемы на материнской плате. Выпускаемые в настоящее время модули SIMM и DIMM имеют разную емкость, но устанавливаются в разъемы одного размера. Например, в один и тот же 100-контактный разъем можно установить модуль DIMM емкостью 4 M × 32, 16 M × 32 или 32 M × 32. Аналогичным образом, в один и тот же 168-контактный разъем можно установить модуль DIMM емкостью 8 M × 64, 16 M × 64 или 64 M × 72. Такие модули занимают на материнской плате очень мало места и позволяют легко увеличивать объем памяти, ведь их ничего не стоит заменить модулями большей емкости.

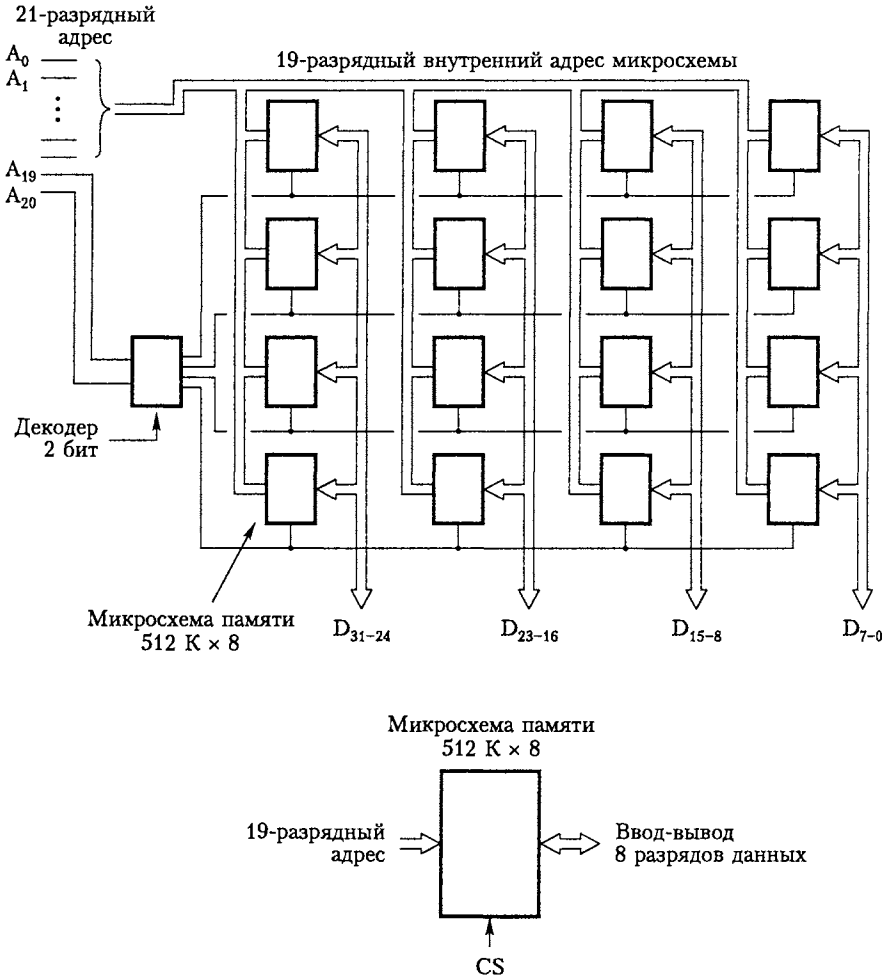


Рис. 5.10. Организация модуля памяти 2 M × 32 на основе микросхем статической памяти 512 K × 8

5.2.6. Замечания относительно системы памяти

При выборе микросхем RAM для конкретной системы учитывают несколько факторов, прежде всего их быстродействие, стоимость, потребляемую мощность и размер.

Статическая RAM обычно используется только в тех случаях, когда на первом месте стоит скорость работы системы. Схемы реализации ее базовых ячеек достаточно сложны, из-за чего стоимость и размер микросхем получаются очень большими. Как правило, статическая RAM применяется для реализации кэш-памяти. Для реализации основной памяти в большинстве компьютеров используется динамическая RAM. Такие микросхемы характеризуются очень высокой плотностью, благодаря чему память достаточно большого объема имеет приемлемую стоимость.

Контроллер памяти

Для сокращения количества внешних контактов адресные входы микросхемы мультиплексируются. Адрес делится на две части. Сначала задаются старшие адресные биты, выбирающие строку массива ячеек. По сигналу RAS эти биты сохраняются в защелках внутри микросхемы, после чего на те же адресные контакты подаются младшие адресные биты, которые сохраняются в защелках по сигналу CAS.

Однако типичный процессор задает весь адрес целиком, одновременно помещая на шину все его разряды. Их мультиплексирование обычно выполняется схемой *контроллера памяти*, расположенной между процессором и динамической памятью (рис. 5.11). В ответ на сигнал запроса, означающий, что необходима операция доступа к памяти, контроллер принимает от процессора полный адрес и сигнал R/$\overline{W}$. Затем контроллер по очереди пересылает в память адреса строки и столбца и генерирует сигналы $\overline{RAS}$ и $\overline{CAS}$. Таким образом, в дополнение к мультиплексированию адреса контроллер осуществляет тактирование RAS и CAS. Он же пересылает в память сигналы R/$\overline{W}$ и CS. Активное состояние сигнала CS обычно соответствует низкому уровню напряжения (на рис. 5.11 он обозначен как $\overline{CS}$). Линии данных непосредственно соединяют процессор и память. Для управления микросхемами SDRAM необходим тактовый сигнал.

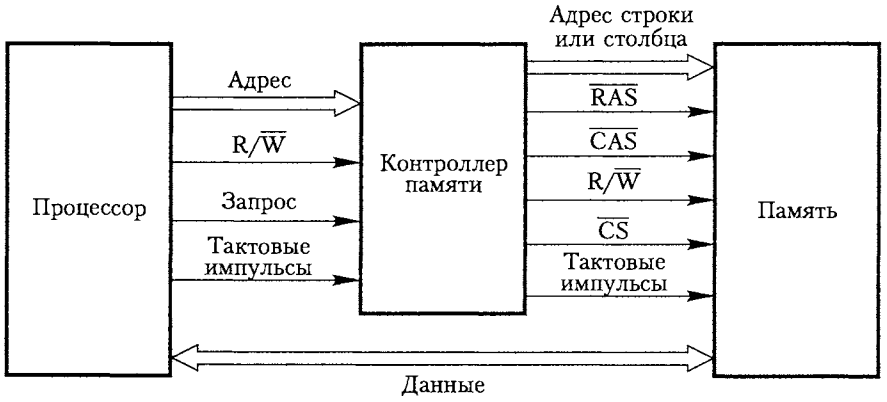


Рис. 5.11. Контроллер памяти

При использовании микросхем DRAM без саморегенерации контроллер памяти отвечает еще и за выдачу сигналов, управляющих процессом регенерации. Он содержит счетчик регенерации, выдающий последовательные адреса строк. Контроллер должен обеспечить необходимую частоту обновления всех строк памяти в соответствии с характеристиками конкретных микросхем.

Издержки на регенерацию

Любая динамическая память нуждается в регенерации. В ранних DRAM время регенерации всех строк составляло порядка 16 мс, но в современных SDRAM оно уже равно 64 мс.

Рассмотрим SDRAM, ячейки которых объединены в строки по 8 К (8192). Предположим, что доступ (чтение) к одной строке занимает четыре такта. Тогда на обновление всех строк уходит $8192 \times 4 = 32768$ тактов. При тактовой частоте 133 МГц обновление всех строк занимает $32768 / (133 \times 10^6) = 246 \times 10^{-6}$ с. Таким образом, на обновление, выполняемое через каждые 64 мс, уходит 0,246 мс времени. А значит, на него затрачивается $0{,}246 / 64 = 0{,}0038$ мс, то есть менее 0,4 % общего времени доступа к памяти.

5.2.7. Память Rambus

Производительность динамической памяти характеризуется временем ожидания и пропускной способностью. Поскольку массивы ячеек во всех микросхемах динамической памяти организованы примерно одинаково, то при использовании одной и той же технологии производства микросхемы имеют практически одинаковое время ожидания. Что касается пропускной способности, то она зависит не только от структуры микросхемы, но и от особенностей соединения таковой с процессором. DDR SDRAM и стандартные SDRAM подключаются к шине процессора. Поэтому скорость пересылки данных зависит не только от быстродействия микросхем памяти, но от пропускной способности шины. Шина с тактовой частотой 133 МГц позволяет через каждые 7,5 нс производить либо одну пересылку, либо две, если задействуются оба фронта тактового сигнала. Единственным способом увеличения количества данных, пересылаемых по шине с ограниченной пропускной способностью линии, является использование большего количества линий, то есть расширение шины.

Однако слишком широкие шины очень дороги и занимают много места на материнской плате. Поэтому в качестве альтернативы можно создать узкую, но более быструю шину. Этот подход используется компанией Rambus Inc., разработавшей собственную шинную архитектуру под названием Rambus. Важнейшим отличием технологии Rambus является особый метод быстрой сигнализации, применяемый в процессе пересылки информации между микросхемами. Вместо сигналов с уровнями напряжения 0 и V_{supply}, представляющими логические значения, в технологии Rambus используются сигналы, характеризующиеся много меньшим отклонением напряжения от базового напряжения V_{ref}. Базовое напряжение составляет около 2 В, а два логических значения задаются напряжениями на 0,3 В выше и ниже базового. Такая система сигналов называется *дифференциальной*. Небольшие отклонения напряжения позволяют сократить время транзакции и ускорить пересылку данных.

При использовании дифференциальной сигнальной системы с высокой скоростью пересылки данных необходима особая технология реализации коммуникационных линий, затрудняющая создание широких шин. Кроме того, для работы с дифференциальными сигналами необходимы особые интерфейсные схемы. Rambus предоставляет полную спецификацию конструкции таких коммуникационных линий, называемых *каналом Rambus*. Современные шины Rambus могут работать на тактовой частоте до 400 МГц, причем данные пересылаются по обоим фронтам тактового сигнала, так что общая частота передачи составляет 800 МГц.

Микросхемы памяти Rambus имеют конструктивные особенности, хотя проектируются на основе стандартной технологии DRAM. Они объединяются в несколько банков памяти, используемых для параллельного доступа к нескольким словам. В состав микросхемы входит интерфейсная схема для шины Rambus. Такие микросхемы называются *Rambus DRAM* (RDRAM).

В исходной спецификации шины Rambus определяется канал из девяти линий данных и множества управляющих линий и линий питания. Восемь линий данных предназначены для пересылки одного байта данных, а девятая — для контроля четности. Последующие спецификации предоставили возможность использовать дополнительные каналы. Двухканальная Rambus, известная также как *Direct RDRAM*, имеет 18 линий данных, что позволяет пересылать по 2 байта данных за раз. Отдельных адресных линий шина не имеет.

Взаимодействие между процессором или другим устройством, играющим роль *хозяина шины*, и модулями RDRAM, являющимися подчиненными устройствами, осуществляется посредством *пакетов*, пересылаемых по линиям данных. Существует три типа пакетов: запроса, подтверждения и данных. Пакет запроса отсылается хозяином шины, и в нем указывается тип выполняемой операции. Этот пакет содержит адрес памяти и 8-разрядный счетчик, определяющий количество пересылаемых байтов. Тип операции задает операции чтения данных из памяти или записи данных в память, а также чтения или записи различных управляющих регистров микросхемы RDRAM. Когда хозяин шины высылает пакет запроса, адресуемое подчиненное устройство отвечает пакетом подтверждения — положительного, если оно может немедленно удовлетворить запрос, или отрицательного в противном случае (тогда хозяин шины должен предпринять новую попытку).

Количество битов в пакете запроса превышает количество линий данных, поэтому на его пересылку требуется несколько тактов. Однако малая ширина коммуникационного соединения компенсируется высокой скоростью пересылки данных.

Микросхемы RDRAM можно объединять в большие модули, подобные SIMM и DIMM. Один такой модуль, называемый RIMM, может содержать до 16 микросхем RDRAM.

Технология Rambus конкурирует с технологией DDR SDRAM. У каждой из них имеются определенные преимущества и недостатки. По причинам нетехнического характера спецификация DDR SDRAM является открытым стандартом, тогда как спецификация RDRAM считается собственностью компании Rambus Inc., и производители микросхем должны за нее платить. На рынке памяти при равной производительности микросхем решающим фактором обычно является цена.

5.3. Память, доступная только для чтения

Микросхемы SRAM и DRAM являются энергозависимыми, и как только питание выключается, хранящаяся в них информация попросту исчезает. Однако существует множество устройств и компонентов, которым требуются запоминающие устройства, сохраняющие информацию и после выключения питания. Примерами таких устройств могут служить жесткие диски обыкновенных компьютеров, предназначенные для хранения огромных объемов информации, в том числе и программного обеспечения операционной системы. Когда компьютер включается, программное обеспечение загружается с диска в основную память. Эту работу выполняет специальная загрузочная программа. Поскольку код загрузочной программы достаточно велик, большая его часть также хранится на диске. Процессор выполняет специальные команды, загружающие программу в память. Если бы вся память состояла только из энергозависимых микросхем, процессор не смог бы получить доступ к этим командам. Поэтому компьютер обычно содержит небольшую энергонезависимую память, в которой хранятся команды, выполняемые при включении компьютера первыми и обеспечивающие копирование программы загрузки с диска в основную память.

Энергонезависимая память особенно часто используется во встроенных системах, о которых рассказывается в главе 9. В таких системах обычно не бывает дисковых запоминающих устройств, и их программы хранятся в энергонезависимой полупроводниковой памяти.

Существуют разные типы энергонезависимой памяти. Как правило, содержимое памяти считывается так же, как из SRAM и DRAM, а вот для его записи применяется специальная процедура. В рабочем режиме содержимое такой памяти только считывается, поэтому она называется *памятью, доступной только для чтения* (Read Only Memory, ROM).

5.3.1. ROM

На рис. 5.12 показана одна из возможных конфигураций ячейки ROM. Логическое значение 0 хранится в ячейке в том случае, если в точке *P* транзистор соединен с «землей»; в противном случае в ней хранится 1. Линия бита через резистор соединена с источником питания. Для того чтобы прочитать информацию о состоянии ячейки, нужно активизировать линию слова. При этом транзисторный ключ закрывается и, если есть соединение между транзистором и «землей», напряжение на линии бита падает почти до нуля. Если соединения с «землей» нет, на линии бита остается высокое напряжение, соответствующее логической единице. Схема считывания на конце линии бита генерирует правильное выходное значение. Данные записываются в ROM при ее производстве.

5.3.2. PROM

Некоторые микросхемы ROM разрабатываются таким образом, что данные в них может записывать пользователь. В этом случае память называется *программируемой ROM* (Programmable ROM, PROM). Для программирования микросхемы

PROM, то есть для записи в нее данных, используется плавкое соединение (точка *P* на рис. 5.12). До программирования во всех ячейках памяти хранятся нули. Для того чтобы поместить в нужные ячейки единицы, пользователь может пережечь плавкие соединения с помощью импульсов усиленного тока. Совершенно очевидно, что этот процесс необратим.

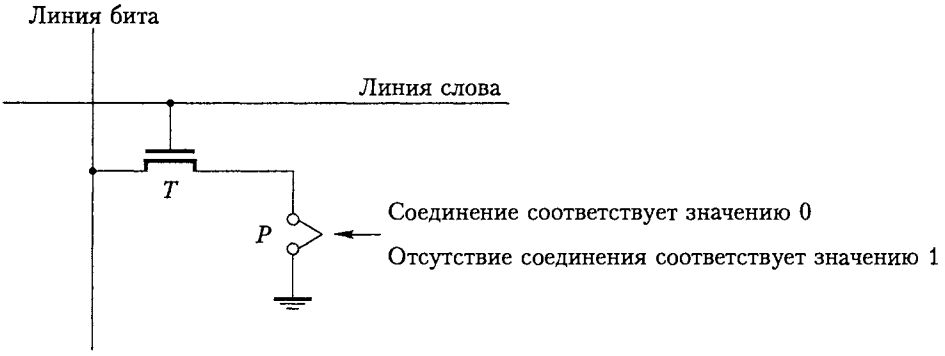


Рис. 5.12. Ячейка PROM

Память PROM гибче и удобнее по сравнению с ROM. Последняя используется в основном для хранения неизменяемых (постоянных) программ и данных, особенно в тех случаях, когда выпускается большое количество одинаковых микросхем. Сравнительно высокая стоимость процесса подготовки шаблона для записи информации в ROM делает производство небольших партий таких микросхем слишком дорогим. В подобных случаях гораздо удобнее и дешевле использовать программируемые пользователем микросхемы PROM.

5.3.3. EPROM

Еще один тип микросхем ROM позволяет не только записывать, но и перезаписывать данные. Такая память обычно называется *стираемой перепрограммируемой ROM* (Erasable Programmable ROM, EPROM). Поскольку EPROM способна хранить информацию в течение длительного периода, она может использоваться вместо ROM для хранения программного обеспечения, которое время от времени должно обновляться.

Структура ячейки EPROM подобна структуре ячейки ROM, показанной на рис. 5.12. Однако в ней всегда имеется соединение с «землей» и используется особый транзистор, который может функционировать или как обычный, или как выключенный. Этот транзистор можно запрограммировать, чтобы он работал как постоянно открытый ключ, поместив в него заряд, который он «захватывает» и не выпускает наружу. Таким образом, ячейка EPROM может использоваться для создания памяти с такой же структурой, как у описанной выше ROM.

Важным преимуществом микросхемы EPROM является то, что ее содержимое можно вытереть и повторно запрограммировать. Для стирания необходимо с помощью ультрафиолетового света удалить заряды, заключенные в транзисторах

ячеек памяти. Поэтому микросхемы EPROM монтируются в непрозрачные корпуса с прозрачными окошками.

5.3.4. EEPROM

У памяти EPROM имеется два существенных недостатка: во-первых, для перепрограммирования чип нужно извлекать из схемы, а во-вторых, при перепрограммировании ультрафиолетовый свет стирает все его содержимое. Существует другая разновидность стираемой программируемой ROM, для которой обе операции можно выполнить электрическим путем. Такие микросхемы, называемые *электронно-перепрограммируемой постоянной памятью* (Electrically Erasable Programmable ROM, EEPROM), для стирания или перезаписи, как вы, по-видимому, уже догадались, не нужно извлекать из компьютера. Более того, их содержимое можно изменять выборочно. Единственным недостатком EEPROM является то, что для стирания, записи и чтения данных в них требуется разное напряжение.

5.3.5. Флэш-память

Одна из сравнительно недавних разработок памяти, подобных EEPROM, получила название *флэш-памяти*. Ячейка такой памяти содержит подобно ячейке EEPROM один транзистор, управляемый «захваченным» зарядом. Однако технологии флэш-памяти и EEPROM, несмотря на большое сходство, существенно различаются: EEPROM позволяет считывать и записывать содержимое одной ячейки, тогда как флэш-память дает возможность считывать ячейки по одной, а записывать только блоками. Перед записью исходное содержимое блока ячеек стирается. Флэш-память имеет большую плотность ячеек, а следовательно, большую емкость и меньшую стоимость в пересчете на бит. Для нее достаточно напряжения питания одного уровня, и к тому же она более экономична.

Благодаря своей экономичности флэш-память удобна для использования в портативных системах, работающих на батареях. В частности она применяется в портативных компьютерах, сотовых телефонах, цифровых видеокамерах и MP3-плеерах. В случае применения в портативных компьютерах и сотовых телефонах флэш-память содержит программное обеспечение, заменяя собой дисковые устройства. В цифровых камерах она используется для хранения изображений, а в MP3-плеерах — для хранения звука. Электронные компоненты схем сотовых телефонов, цифровых камер и MP3-плееров могут служить примерами встроенных систем, о которых рассказывается в главе 9.

Возможно, вы знаете, что для описанных выше систем недостаточно емкости одной микросхемы флэш-памяти. В них используются большие модули, состоящие из множества микросхем. Существует две популярные разновидности таких модулей: флэш-карты и флэш-диски.

Флэш-карты

Для создания большого модуля флэш-микросхемы можно вмонтировать в небольшую карту. Такие карты имеют стандартный интерфейс, благодаря чему их можно использовать в самых разных устройствах. Их емкость может быть разной,

как правило, она составляет 8, 32 или 64 Мбайт. Для хранения одной минуты музыкальной записи в формате MP3 необходимо около 1 Мбайт памяти, так что на карте объемом 64 Мбайт можно записать музыку, которая будет воспроизводиться примерно в течение 1 ч. Вмонтировать флэш-карту в устройство очень легко: она просто вставляется в удобный слот.

Флэш-диски

Существуют и более крупные модули флэш-памяти, предназначенные для замены жестких дисков. Каждый такой модуль полностью эмулирует жесткий диск и может вставляться в предназначенный для него отсек. Однако емкость флэш-диска значительно меньше. В настоящее время предельная вместимость флэш-диска составляет менее 1 Гбайт, тогда как обычный жесткий диск может хранить десятки гигабайтов данных.

Важным преимуществом флэш-диска является то, что он представляет собой стационарное электронное устройство, не содержащее подвижных элементов. Время поиска и доступа у него гораздо меньше по сравнению с обычным жестким диском (см. раздел 5.9), а значит, меньше и время ответа. Кроме того, флэш-диск потребляет меньше энергии, что делает его очень удобным компонентом для устройств, работающих от батарей. И, наконец, он устойчив к вибрациям.

Недостатками флэш-диска являются его меньшая емкость и более высокая стоимость в пересчете на бит. Кроме того, после определенного количества операций записи флэш-память разрушается. Впрочем, это количество достаточно велико — каждую ячейку можно перезаписать по меньшей мере миллион раз.

5.4. Быстродействие, объем и стоимость

Мы уже говорили о том, что в идеале память должна быть быстрой, большой и дешевой. Из сказанного в разделе 5.2 ясно, что для реализации очень быстрой памяти лучше всего использовать микросхемы SRAM. Однако они довольно дороги, поскольку каждая ячейка такой памяти содержит шесть транзисторов, из-за чего на одной микросхеме невозможно разместить много ячеек. Таким образом, большую память на основе микросхем SRAM нецелесообразно создавать из чисто экономических соображений. Для нее применяются гораздо более дешевые микросхемы динамической RAM с очень простыми ячейками. Правда, работает такая память медленнее.

Хотя в компьютер за вполне разумную цену можно установить сотни мегабайтов динамической памяти, этого все равно недостаточно для современных программ, обрабатывающих огромные объемы данных. Поэтому для увеличения адресного пространства памяти используются внешние запоминающие устройства, прежде всего магнитные диски. Современные диски имеют очень большую емкость и, как правило, приемлемую цену, поэтому они достаточно интенсивно применяются в компьютерных системах. Однако функционируют магнитные диски значительно медленнее полупроводниковой памяти. Из всего сказанного следует такой вывод: очень большой объем памяти за приемлемую цену можно получить только в виде жесткого диска. Для реализации основной памяти большого объема

подходит технология RAM. А микросхемы SRAM можно использовать для создания очень быстрых запоминающих устройств небольшого объема, таких как кэш-память.

Как правило, в компьютере используются все три типа памяти. Таким образом, систему памяти компьютера можно представить в виде иерархии, показанной на рис. 5.13. Быстрее всего осуществляется доступ к данным, хранящимся в регистрах процессора. Поэтому, если рассматривать эти регистры как составляющую иерархии памяти, то с точки зрения быстродействия они располагаются на самом верху, хотя по объему составляют ничтожно малую часть всей памяти компьютера.

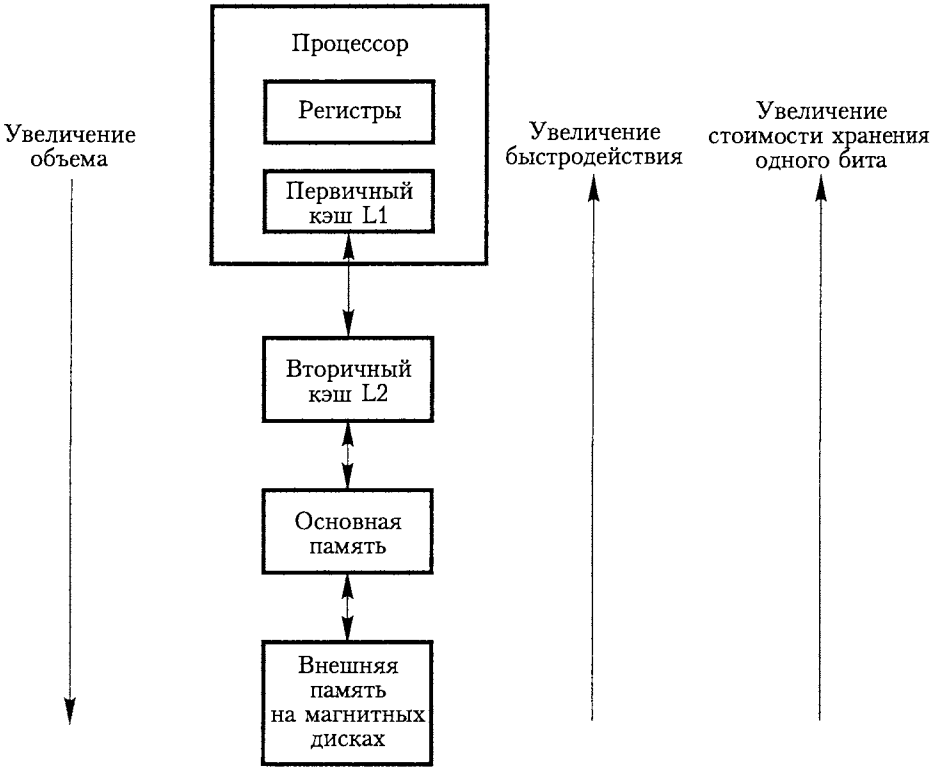


Рис. 5.13. Иерархия памяти

На следующем уровне иерархии располагается сравнительно небольшой объем памяти, который можно реализовать прямо в микросхеме процессора. Эта память, называемая *кэшем процессора*, содержит копии команд и данных, хранящихся во внешней по отношению к процессору и намного большей основной памяти. Идея кэш-памяти, проиллюстрированная на рис. 1.6, подробно рассматривается в разделе 5.5. Обычно в компьютере имеется два уровня кэш-памяти. Первичный кэш располагается на микросхеме процессора и называется *кэшем первого уровня* (L1). Вторичный кэш имеет больший объем, располагается между

первичным кэшем и остальной памятью и называется *кэшем второго уровня* (L2). Для его реализации обычно используются микросхемы SRAM.

Типичный компьютер содержит кэш первого уровня, располагаемый на микросхеме процессора, и внешний по отношению к процессору кэш второго уровня, несколько большего объема. Однако это не всегда так. Бывает, что микросхема процессора вообще не содержит кэша или же, напротив, содержит кэш обоих уровней.

Ниже по иерархии располагается *основная память*. Она довольно велика и реализуется на основе микросхем динамической памяти, как правило, в виде модулей SIMM, DIMM и RIMM. Основная память значительно больше и намного медленнее кэша. В типичном компьютере время доступа к основной памяти в десять раз меньше времени доступа к кэшу L1.

Дисковые устройства предоставляют, можно сказать, огромный объем недорогой памяти, но по сравнению с полупроводниковыми устройствами они очень медленные. Более подробно мы поговорим о них в разделе 5.9.

Для выполнения программ исключительное значение имеет скорость доступа к памяти. Идея управления иерархической системой памяти состоит в том, чтобы переместить команды и данные, которые будут использоваться в ближайшее время, как можно ближе к процессору. Для этого применяются специальные механизмы, описанные в следующих разделах. Их анализ мы начнем с кэш-памяти.

5.5. Кэш-память

По сравнению с быстродействием современных процессоров скорость функционирования основной памяти очень мала. Однако процессор не может тратить много времени в ожидании команд и данных из основной памяти. Поэтому нужны механизмы, сокращающие время доступа к необходимой информации. Поскольку быстродействие основной памяти физически ограничено, здесь требуется архитектурное решение. Таким решением является использование быстрой *кэш-памяти*, благодаря которой основная память представляется процессору более быстрой, чем есть на самом деле.

Эффективность механизма кэширования основывается на свойстве компьютерных программ, называемом *локализацией ссылок*. Анализ процесса реализации различных программ показывает, что большую часть времени в них выполняется код, в котором определенные группы команд повторяются по многу раз. Это простые и вложенные циклы, а также многократно вызываемые подпрограммы. Причем последовательность выполнения команд не имеет значения — важно то, что многие из них в локализированных областях программы многократно повторяются в течение определенного промежутка времени, а доступ к оставшейся части программы осуществляется сравнительно редко. Это и называется локализацией ссылок. Локализация ссылок происходит и во времени, и в пространстве. Локализация во времени означает, что недавно выполнявшиеся команды, скорее всего, очень скоро будут выполнены снова. А локализация в пространстве означает большую вероятность того, что очень скоро будут выполнены команды, расположенные в непосредственной близости от только что реализованных команд (имеется в виду близость адресов команд).

Если поместить активные сегменты программы в быструю кэш-память, общее время их выполнения значительно сократится. Идея кэширования команд очень проста. Управляющие схемы памяти разрабатываются таким образом, чтобы можно было использовать свойство локализации ссылок. Исходя из принципа локализации во времени, каждый элемент, к которому обращается процессор, будь то команда программы или элемент данных, копируется в кэш, где он остается до тех пор, пока не потребуется снова. Исходя из принципа локализации в пространстве, в кэш копируется не только текущий элемент программы или данных, но еще и несколько близлежащих элементов. Набор элементов с последовательными адресами определенного размера мы будем называть *блоком* или *строкой кэша*.

Обратимся к простой схеме, показанной на рис. 5.14. Если процессор выдает запрос чтения, содержимое блока памяти считывается по заданному адресу по одному слову в кэш. Когда впоследствии программа обратится к любому элементу этого блока, он будет прочитан не с диска, а прямо из кэша. Обычно в каждый конкретный момент времени в кэш-памяти может храниться достаточно много блоков, но по сравнению с их общим количеством в основной памяти это очень мало. Соответствие между блоками в основной памяти и блоками в кэше определяется *функцией отображения*. Когда кэш полон и производится обращение к отсутствующему в нем слову памяти (команде или данным), управляющее кэшем аппаратное обеспечение должно решить, какой из блоков удалить из кэша, чтобы добавить в него новый блок, содержащий требуемое слово. Набор правил для принятия такого решения составляет *алгоритм замещения*.

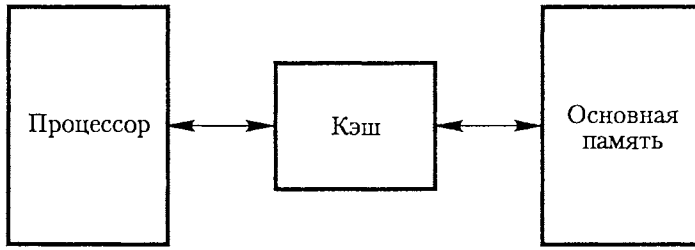


Рис. 5.14. Использование кэш-памяти

Процессор ничего не знает о существовании кэша. Он просто выдает запросы чтения и записи, используя адреса, которые указывают на память. В ответ схема управления кэшем выясняет, имеется ли в таковом запрошенное слово. Если да, то в операции чтения или записи задействуется слово из кэша. При этом говорят, что имеет место *попадание в кэш*. В случае операции считывания обращение к основной памяти вообще не происходит. Если же выполняется операция записи, система может действовать одним из двух способов. При использовании первого из них, называемого протоколом *сквозной записи*, предполагается, что кэш и основная память обновляются одновременно. Второй способ подразумевает, что данные обновляются только в кэше, после чего помечаются с помощью соответствующего битового флага. Упомянутый бит называют флагом изменения или модификации. Соответствующее слово в основной памяти обновляется позже, при

удалении из кэша того блока, который содержит помеченное слово. Описанная технология называется протоколом *обратной записи* или *обратного копирования*. Протокол сквозной записи проще, но при его использовании производятся лишние операции записи в основную память — в том случае, если некоторое слово обновляется в кэше несколько раз подряд. Конечно, подобное происходит и при обратной записи, поскольку при удалении блока из кэша в память записываются все его слова, даже если изменилось только одно из них.

Ситуация, при которой слово, адресуемое операцией считывания, отсутствует в кэше, называется *промахом чтения*. В этом случае из основной памяти в кэш копируется блок, содержащий такое слово. Запрошенное слово передается в процессор после загрузки в кэш всего блока. В качестве альтернативы оно может быть передано в процессор сразу после его прочтения из основной памяти. Последний подход, называемый *сквозной загрузкой*, сокращает время ожидания слова процессором, но для его реализации требуется более сложная схема.

Если в кэше не оказывается слова, адресуемого операцией записи, мы говорим, что возникла ситуация, называемая *промахом записи*. В таком случае, при использовании протокола сквозной записи, информация записывается прямо в основную память. Но если применяется протокол обратной записи, в кэш сначала копируется блок, содержащий заданное слово, а затем это слово перезаписывается в кэше и помечается как измененное.

5.5.1. Функция отображения

Чтобы показать, как происходит выборка блоков данных из памяти в кэш, мы обратимся к простому примеру. Предположим, у нас имеется кэш, состоящий из 128 блоков по 16 слов в каждом, то есть с общим количеством слов, равным 2048 (2 К), и основная память, адресуемая с помощь 16-разрядных адресов. Основная память имеет объем 64 К слов, который мы будем рассматривать как 4 К блоков по 16 слов. Чтобы облегчить восприятие излагаемого материала, будем считать, что последовательные адреса указывают на последовательные слова.

Прямое отображение

Простейшим способом сопоставления адресов блоков в кэше и в памяти является *прямое отображение*. При использовании этой технологии блок j основной памяти отображается на блок j по модулю 128 кэша, как показано на рис. 5.15. Таким образом, когда загружается один из блоков основной памяти, начинающихся по адресам 0, 128, 256 и т. д., он записывается в блок кэша 0. Блоки 1, 129, 257 и т. д. записываются в блок кэша 1 и т. д. Поскольку на каждый блок кэша отображается более одного блока основной памяти, то даже при не до конца заполненном кэше может возникнуть состязание за некоторую позицию. Например, команды программы, начавшиеся в блоке 1, после перехода могут продолжиться в блоке 129. В результате выполнения программы оба эти блока должны быть скопированы в блок 1 кэша. Конфликт разрешается просто: старый блок заменяется новым.

Таким образом, местоположение блока в кэше определяется на основе его адреса в памяти. Адрес в памяти может быть разделен на три поля (рис. 5.15). Четыре младших разряда задают одно из 16 слов блока. Когда в кэш записывается новый

блок, 7-разрядное поле номера блока данного кэша определяет его местоположение. Пять старших разрядов задают адрес блока в памяти. При записи блока в кэш они указываются в специальном поле тега (дескрипторе). Такое поле имеется в кэше для каждого из его блоков — оно определяет, какому из 32 блоков памяти, отображаемых на данный блок кэша, соответствует хранящаяся здесь информация. Во время выполнения программы процессор генерирует адреса, в каждом из которых 7-разрядное поле номера блока кэша указывает на конкретный блок в нем. Тег этого блока сравнивается со старшими 5 разрядами адреса, и если они совпадают, значит, данное слово уже находится в кэше. В противном случае блок, содержащий данное слово, нужно извлечь из основной памяти и поместить в кэш. Технология прямого отображения очень проста, но ей недостает гибкости.

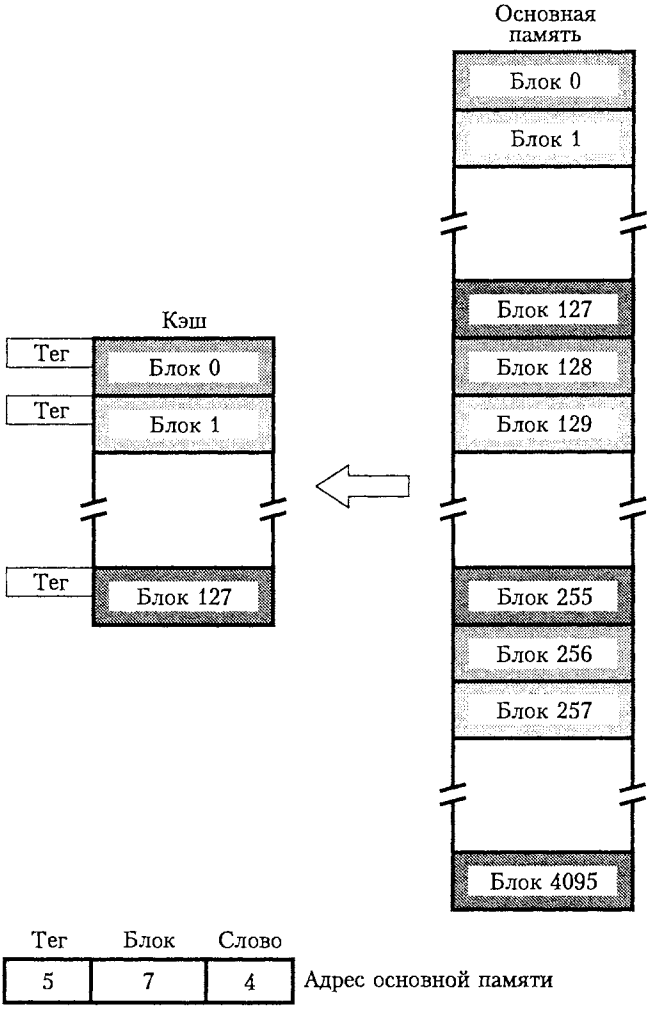


Рис. 5.15. Кэш с прямым отображением

Ассоциативное отображение

На рис. 5.16 показана схема гораздо более гибкого метода отображения, согласно которому блок основной памяти можно помещать в любой блок кэша. При этом для идентификации хранящегося в кэше блока памяти необходимо иметь уже не 5, а 12 бит. При выполнении программы теговые биты сгенерированного процессором адреса по очереди сравниваются с теговыми битами каждого блока кэша. Если совпадение найдено, значит, содержащий данное слово блок уже присутствует в кэше. Такая технология называется *ассоциативным отображением*. Она предоставляет полную свободу выбора местоположения блока в кэше, благодаря чему пространство кэша может использоваться более эффективно. Новые блоки заменяют уже хранящиеся в кэше только в том случае, если кэш заполнен, причем для этой цели необходим алгоритм выбора удаляемого блока. Подобных алгоритмов, как станет ясно из раздела 5.5.2, довольно много. Стоимость ассоциативного кэша выше, чем кэша с прямым отображением, поскольку в нем выполняется просмотр всех 128 тегов блоков. Поиск блока в кэше называется *ассоциативным поиском*. Для того чтобы он выполнялся достаточно быстро, теги должны просматриваться параллельно.

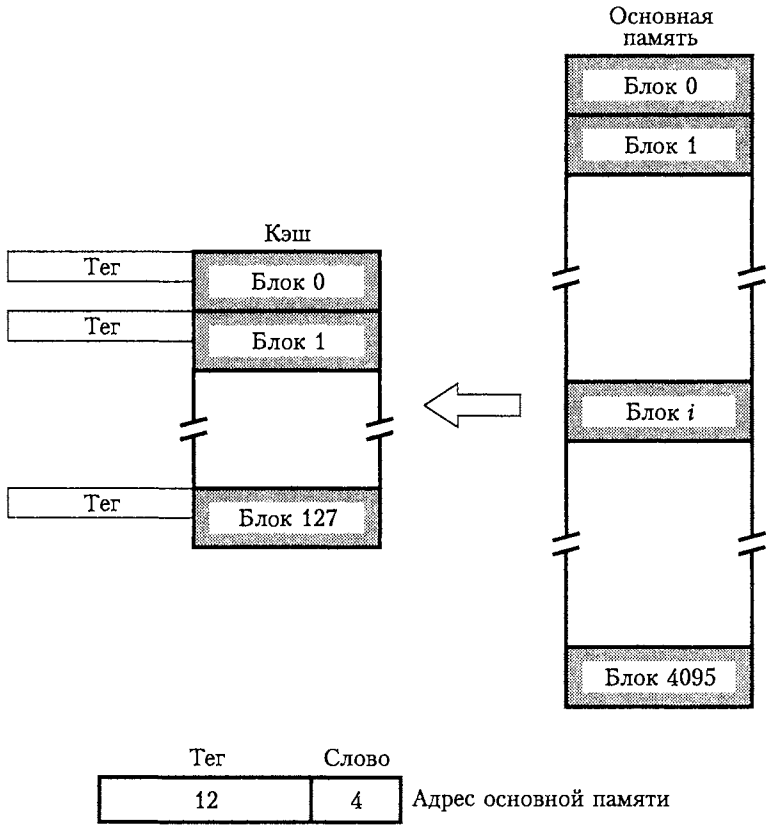


Рис. 5.16. Ассоциативный кэш

Множественно-ассоциативное отображение

Технологии прямого и ассоциативного отображения могут использоваться совместно. В этом случае блоки кэша объединяются в множества, и каждый блок основной памяти может располагаться в любом из блоков определенного множества. Причем вероятность конфликтов, являющихся одним из недостатков прямого отображения, значительно снижается. Такой кэш, получивший название *множественно-ассоциативного*, дешевле полностью ассоциативного кэша, поскольку в нем уменьшена область ассоциативного поиска. Рассмотрим принцип множественно-ассоциативного отображения на примере кэша с 64 множествами по два блока в каждом (рис. 5.17). Блоки памяти 0, 64, 128, ..., 4032 отображаются на множество 0 и могут занимать любую из двух позиций в этом множестве. Наличие 64 множеств блоков означает, что 6-разрядное поле множества в составе адреса слова определяет, какое множество кэша может содержать это слово. Поле тега адреса ассоциативным путем сравнивается с тегами двух блоков найденного множества, и если оно совпадет с одним из тегов, значит, соответствующий блок уже находится в кэше. Реализовать такой поиск очень просто.

Количество блоков во множестве задается в соответствии с требованиями конкретного компьютера. В случае основной памяти и кэша, показанных на рис. 5.17, для четырех блоков в множестве потребуется 5-разрядное поле множества, для восьми блоков — 4-разрядное и т. д. Граничное значение 128 блоков в множестве не требует поля множества и соответствует полностью ассоциативному кэшу с 12 теговыми битами. Другое граничное значение — один блок в множестве — соответствует методу прямого отображения. Кэш с k блоками во множестве называется k-канальным множественно-ассоциативным кэшем.

Для каждого блока в кэше должен храниться еще один управляющий бит, называемый битом достоверности. Он указывает, содержит ли блок достоверные данные. Его не следует путать с упоминавшимся ранее битом изменения, указывающим, был ли блок модифицирован за то время, пока он находится в кэше. Бит модификации нужен только в тех системах, в которых не используется сквозная запись. При включении питания системы и при загрузке с диска в основную память новой программы и данных все биты достоверности устанавливаются в 0. Пересылка данных между диском и основной памятью управляется механизмом прямого доступа к памяти (DMA). Обычно эти данные минуют кэш, что вызвано соображениями стоимости и производительности. Когда блок кэша в первый раз загружается из основной памяти, его бит достоверности устанавливается в 1. Если блок основной памяти обновляется из другого источника, минуя кэш, система проверяет, находится ли загружаемый блок в кэше. Если да, его бит достоверности устанавливается в 0, чтобы в кэше не оказалось *устаревших* данных.

Подобная проблема возникает при ПДП-пересылке данных из основной памяти на диск, если используется кэш с обратной записью. Данные, находящиеся в памяти, могут не отражать изменений, внесенных в кэшируемую копию. Поэтому перед их копированием на диск можно записать измененные данные из кэша в основную память. Операционная система легко справляется с этой задачей, и это не отражается на ее производительности, поскольку пересылка данных между диском и основной памятью происходит нечасто. Обязательное использование двумя

разными элементами (в данном случае процессором и подсистемой ПДП) одинаковых копий данных называется *согласованностью кэша*.

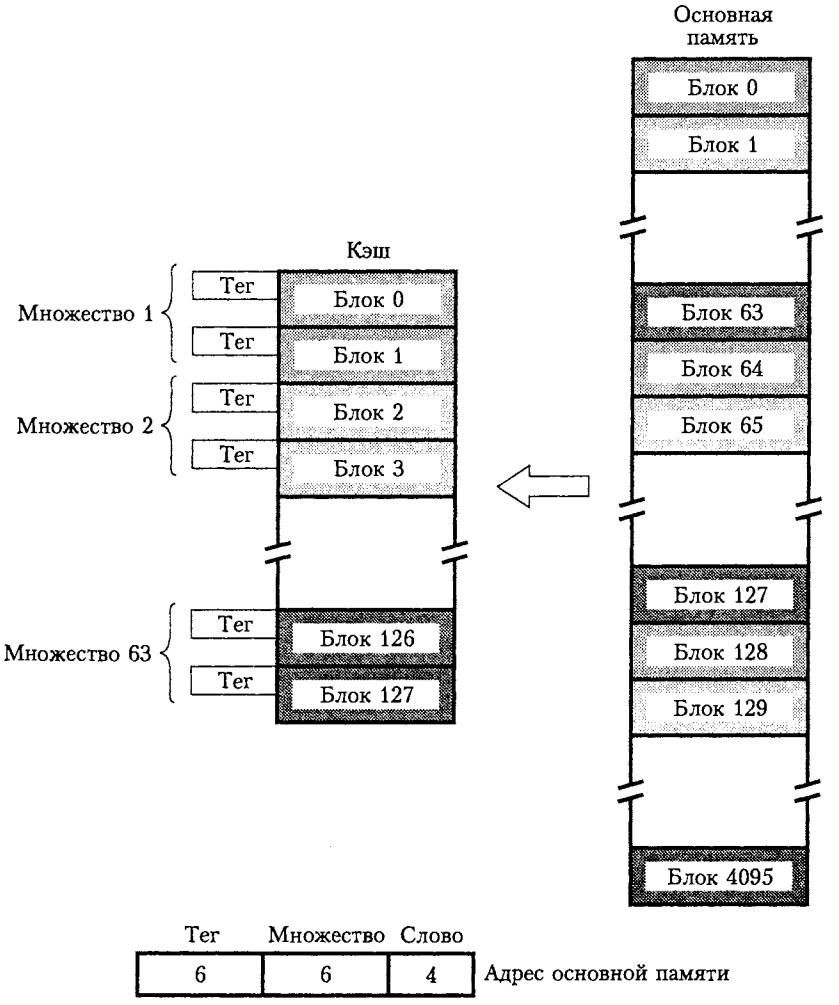


Рис. 5.17. Множественно-ассоциативный кэш с двумя блоками в множестве

5.5.2. Алгоритмы замещения

В кэше с прямым отображением позиция каждого блока определена раз и навсегда, поэтому никакая особая стратегия замены блоков ему не требуется. А вот в ассоциативном или множественно-ассоциативном кэше замена блоков может выполняться по-разному. Когда в кэш нужно будет поместить новый блок, но свободной позиции для него там не окажется, контроллер кэша должен выбрать один из старых блоков для перезаписи. От того, как он будет решать эту задачу, зависит производительность системы. Главная идея, которой следует руководствоваться,

принимая такое решение, состоит в следующем: в памяти должны оставаться блоки, для которых вероятность того, что они понадобятся в ближайшем будущем, максимальна. Но как их определить? Здесь можно опереться на принцип локализации ссылок. Так как повторяющиеся команды, лежащие в пределах некоторой области, выполняются в течение определенного времени, существует большая вероятность того, что блоки, обращение к которым производилось недавно, очень скоро потребуются снова. Поэтому, когда требуется освободить место для нового блока, имеет смысл удалить из кэша тот блок, к которому дольше всего не было обращений. Алгоритм работы этого блока называется алгоритмом удаления наиболее давно использовавшихся элементов (Least Recently Used, LRU).

Для использования алгоритма LRU контроллер кэша должен отслеживать обращения ко всем блокам кэша. Предположим, что ему нужно следить за обращениями к блоку LRU из четырехблочного множества множественно-ассоциативного кэша. Для каждого блока может использоваться 2-разрядный счетчик. При попадании в кэш счетчик соответствующего блока устанавливается в 0. Счетчики, значения которых были больше значения данного счетчика, увеличиваются на 1. Когда в кэше не оказывается нужного блока, а в множестве еще есть место, счетчик нового блока устанавливается в 0, а значения других счетчиков увеличиваются на 1. Если же множество заполнено, блок, счетчик которого равен 3, удаляется, а на его место помещается новый блок. Значения остальных трех счетчиков увеличиваются на 1. Нетрудно убедиться, что при использовании такого алгоритма значения счетчиков четырех блоков всегда будут разными.

Алгоритм LRU очень популярен. В большинстве случаев он работает прекрасно, но иногда его применение может привести к снижению производительности, например, при обращении к последовательным элементам массива, который слишком велик и не помещается в кэше целиком (см. раздел 5.5.3 и упражнение 5.12). Для того чтобы повысить производительность алгоритма, можно внести в него некоторую долю случайного выбора.

На практике используются и некоторые другие алгоритмы замещения. Правило замены самого «старого» блока кажется наиболее логичным, но оно не принимает в расчет частоту обращений к хранящимся в кэше блокам. Поэтому оно не так эффективно, как алгоритм LRU. Самым простым решением является случайный выбор перезаписываемого блока, и, что интересно, практика показывает его эффективность.

5.5.3. Примеры технологий отображения

Ниже будет рассмотрен пример, демонстрирующий различия между разными технологиями отображения памяти на кэш. Предположим, что у процессора имеются отдельные кэши команд и данных. Для упрощения примера будем считать, что в кэше данных помещается только восемь блоков. Блок состоит из одного 16-разрядного слова, а память адресуется пословно посредством 16-разрядных адресов. (Это не реалистичные параметры, но они удобны для нашего примера.) Для замены блоков в кэше используется алгоритм LRU.

Давайте проанализируем изменения в кэше данных, вызванные выполнением следующей задачи. Массив чисел A размером 4 × 14, в котором каждое число

занимает одно слово, хранится в основной памяти по шестнадцатеричным адресам от 7A00 до 7A27. Элементы этого массива хранятся в порядке следования столбцов. На рис. 5.18 показано, как выделяются теги из адресов памяти при разных технологиях отображения. Обратите внимание, что в нашем примере нет специальных битов, используемых для идентификации слова внутри блока, как на рис. 5.15–5.17, поскольку мы предполагаем, что каждый блок содержит только одно слово. Приложение нормализует значения элементов первой строки массива A относительно среднего значения элементов этой строки. Таким образом, нам нужно вычислить среднее значение элементов строки и разделить на него значение каждого из элементов. Эту задачу можно выразить так:

$$A(0,i) \leftarrow \frac{A(0,i)}{\left(\sum_{j=0}^{9} A(0,j)\right)/10} \quad \text{для } i = 0, 1, \ldots, 9$$

Код, выполняющий указанную задачу, приведен на рис. 5.19. В программе на машинном языке для ссылки на элементы массива будут использоваться адреса памяти. Для хранения суммы и среднего значения предназначены переменные SUM и AVE. Эти переменные, равно как и индексные переменные i и j, при вычислениях хранятся в регистрах процессора.

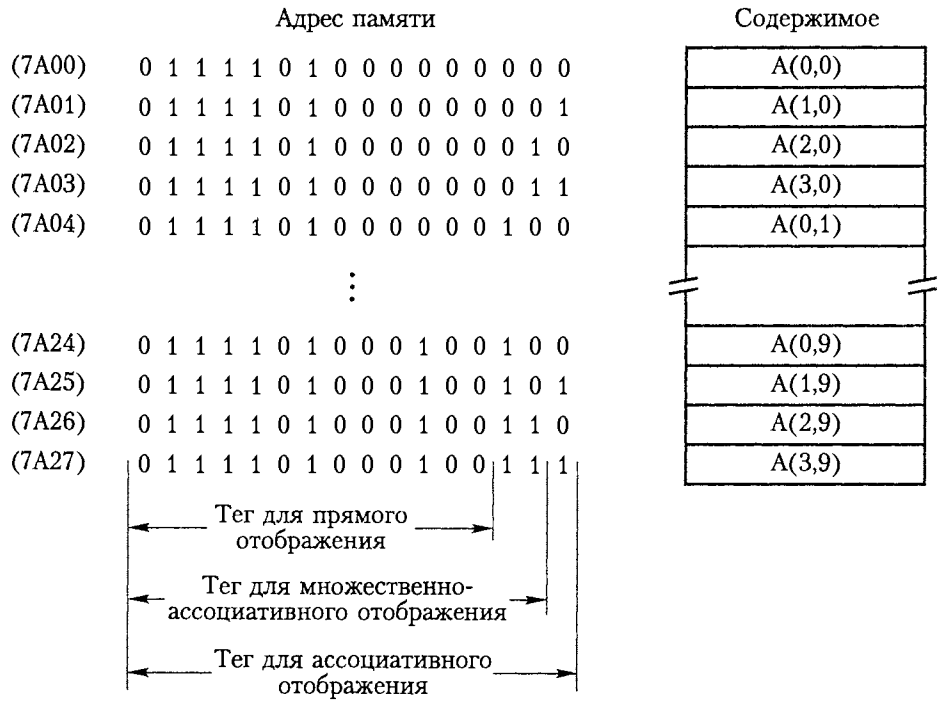


Рис. 5.18. Массив, хранящийся в основной памяти

```
SUM := 0
for j:=0 to 9 do
      SUM := SUM + A(0,j)
end
AVE := SUM / 10
for i := 9 downto 0 do
      A(0,i) := A(0,i) / AVE
end
```

Рис. 5.19. Код для примера из раздела 5.5.3

Кэш с прямым отображением

На рис. 5.20 показано, как изменяется содержимое кэша с прямым отображением. В столбцах таблицы приведено содержимое кэша после проходов по двум циклам программы, представленной на рис. 5.19. Например, после второго прохода по первому циклу ($j = 1$) в кэше содержатся элементы A(0,0) и A(0,1). Они хранятся в блоках 0 и 4 с учетом значений трех младших разрядов их адресов. На следующем проходе элемент A(0,0) заменяется элементом A(0,2), имеющим тот же адрес блока. Обратите внимание, что элементы массива соответствуют только двум блокам кэша, а остальные блоки остаются неизменными — в конце процесса нормализации они содержат те же данные, что до его начала.

Позиция блока	Содержимое кэша данных после прохода по циклу:								
	$j = 1$	j = 3	j = 5	j = 7	j = 9	$i = 6$	$i = 4$	$i = 2$	$i = 0$
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

Рис. 5.20. Содержимое кэша данных с прямым отображением

После десяти проходов по первому циклу ($j = 9$) в кэше хранятся элементы A(0,8) и A(0,9). Поскольку при выполнении второго цикла элементы обрабатываются в обратном порядке, то элементы, необходимые на первых двух проходах этого цикла, будут находиться в кэше. На третьем проходе ($j = 7$) элемент A(0,8) будет заменен элементом A(0,7), затем элементом A(0,6) и т. д. Таким образом, во время выполнения второго цикла в кэше будет заменено восемь элементов.

Читатель должен иметь в виду, что каждому блоку в кэше соответствует определенный тег. На рисунке мы их не показываем, чтобы не занимать лишнего места.

Ассоциативный кэш

Как изменяется содержимое полностью ассоциативного кэша, можно судить по рис. 5.21. Если перед началом выполнения программы кэш пуст, на первых восьми проходах по циклу элементы массива копируются в последовательные позиции. Для эффективного кэширования данных важно, чтобы второй цикл перебирал элементы массива в обратном порядке. Интересно посмотреть, что получится, если второй цикл пройдет по элементам в том же порядке. что и первый (см. упражнение 5.12). Если используется алгоритм LRU, то во втором цикле все элементы будут перезаписаны еще до того, как они будут обработаны. Этого не произойдет, если применить алгоритм замены со случайной выборкой блоков.

	Содержимое кэша данных после выполнения цикла:				
Позиция блока	$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

Рис. 5.21. Содержимое ассоциативного кэша данных

Множественно-ассоциативный кэш

Предположим, что множественно-ассоциативный кэш данных разделен на два множества по четыре блока. Это означает, что младший бит адреса будет определять множество, которому соответствует данный блок памяти. Старшие 15 разрядов составляют тег.

Процесс изменения содержимого кэша показан на рис. 5.22. Поскольку все блоки данных нашей программы в памяти имеют четные адреса, все они отображаются на множество 0. Из-за этого шесть элементов при выполнении второго цикла перезагружаются.

Даже из этих простых примеров видно, что полностью ассоциативное отображение наиболее эффективно, за ним следует множественно-ассоциативное отображение, а наименее эффективно прямое отображение. Однако вследствие того, что реализация полностью ассоциативного отображения слишком дорого стоит, в качестве компромиссного решения, как правило, используется множественно-ассоциативное отображение.

	Содержимое кэша данных после выполнения цикла:					
	j = 3	j = 7	j = 9	i = 4	*i* = 2	i = 0
Множество 1	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
Множество 2						

Рис. 5.22. Содержимое множественно-ассоциативного кэша данных

5.5.4. Организация кэша в коммерческих процессорах

Далее речь пойдет о реализации кэша в процессорах 68040, ARM710T, а также Pentium III и Pentium 4.

Кэш-память процессора 68040

Процессор 68040 компании Motorola содержит два кэша, встроенных прямо в микросхему процессора: один для команд, а другой для данных. Оба кэша имеют емкость 4 Кбайт и четырехканальную множественно-ассоциативную организацию (рис. 5.23). Каждый кэш разделен на 64 множества по 4 блока. Размер блока равен 4 длинным словам, а размер длинного слова — 4 байтам. Для отображения блоков памяти на блоки кэша адреса интерпретируются так, как показано на рисунке. Четыре младших разряда определяют позицию байта в блоке. Следующие 6 разрядов выбирают одно из 64 множеств, а старшие 22 разряда составляют тег. Для компактности содержимое этих полей приведено в шестнадцатеричном формате.

Механизм управления кэшем подразумевает наличие для каждого блока одного бита достоверности, а также одного бита модификации для каждого длинного слова блока. Бит достоверности устанавливается в 1, когда соответствующий блок в первый раз загружается в кэш. С каждым длинным словом связан отдельный бит модификации, который устанавливается в 1, когда данные этого слова изменяются в результате операции записи. Он остается в этом состоянии до тех пор, пока содержимое блока не будет скопировано в основную память.

При обращении к кэшу теговые разряды адреса сравниваются с четырьмя тегами в соответствующем множестве. Если один их них совпадет с адресом и если бит достоверности соответствующего блока равен 1, значит, произошло попадание в кэш. На рис. 5.23 в качестве примера приведен случай, когда адресуемые данные найдены в третьем длинном слове четвертого блока множества 0.

Для кэша данных может быть использована либо обратная, либо сквозная запись — выбор одного из двух протоколов осуществляется операционной системой. Содержимое кэша команд изменяется только в том случае, если новые команды загружаются в результате промаха чтения. Когда новый блок должен быть помещен в заполненное множество, алгоритм замещения выбирает удаляемый блок случайным образом. Если один или несколько битов достоверности указанного блока установлены в 1, перед их удалением выполняется обратная запись.

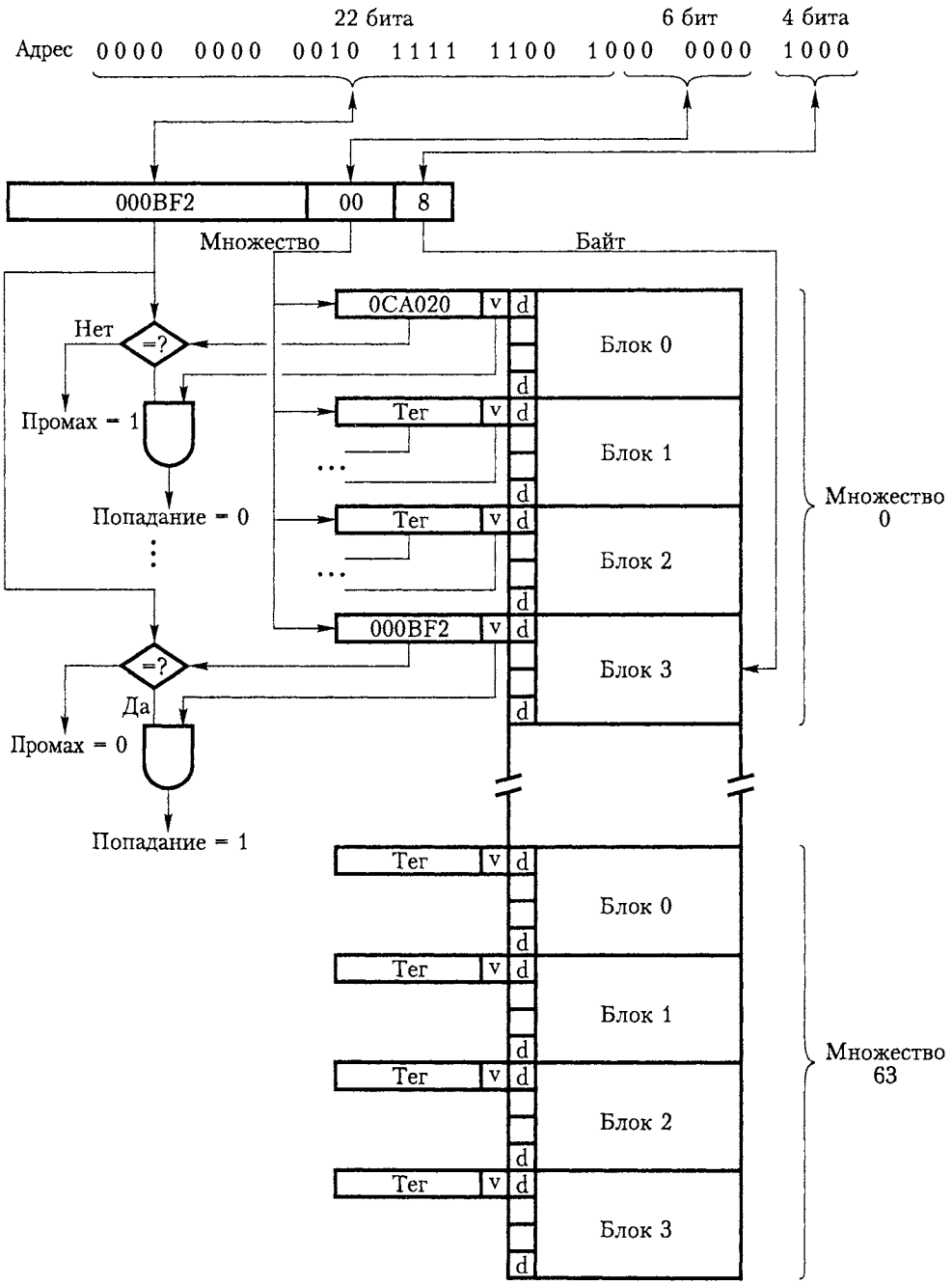


Рис. 5.23. Кэш данных процессора 68040

Кэш процессора ARM710T

Процессоры семейства ARM, имеющие архитектуру типа RISC, отличаются низкой стоимостью и малым потреблением энергии. Один из таких процессоров называется ARM710T. Его единственный кэш предназначен для команд и данных.

Структура кэша процессора ARM710T подобна структуре кэша, показанного на рис. 5.23. Это четырехканальный множественно-ассоциативный кэш с 16-байтовыми блоками, каждый из которых состоит из четырех 32-разрядных слов. Для управления этим кэшем используется протокол со сквозной записью и алгоритм случайного выбора замещаемых блоков.

Кэш процессора ARM710T достаточно полно соответствует требованиям минимальной стоимости и минимального потребления энергии. Ведь проще реализовать единый универсальный кэш для команд и данных, чем два отдельных кэша. Сказанное касается также сквозной записи и случайного выбора замещаемых блоков.

Кэши процессора Pentium III

Pentium III был задуман как процессор с очень высокой производительностью. А поскольку производительность зависит от скорости доступа к командам и данным, он снабжен двухуровневым кэшем. Кэш-память первого уровня состоит из кэша команд объемом 16 Кбайт и такого же кэша данных. Кэш данных имеет четырехканальную множественно-ассоциативную организацию и для него может применяться как обратная, так и сквозная запись. Кэш команд имеет двухканальную множественно-ассоциативную организацию. Поскольку в ходе выполнения программы команды обычно не модифицируются, особая стратегия записи для кэша команд не нужна.

Кэш второго уровня имеет гораздо больший объем. В нем содержатся и команды и данные. Этот кэш соединен с остальной частью системы, как показано на рис. 5.24. Блок шинного интерфейса соединяет все три кэша, основную память и устройства ввода-вывода. Для взаимодействия между процессором и остальными устройствами предназначены две раздельные шины: быстрая шина кэша, соединяющая процессор с кэшем второго уровня, и более медленная системная шина, соединяющая процессор с основной памятью и устройствами ввода-вывода.

Кэш второго уровня может быть реализован вне микросхемы процессора, как в версии процессора Pentium III под названием Katmai. L2-кэш этого процессора имеет размер 512 Кбайт и реализован на основе памяти SRAM. Он имеет четырехканальную множественно-ассоциативную организацию, может использовать протоколы сквозной и обратной записи. Шина данного кэша имеет ширину 64 бита.

Усовершенствование технологий создания СБИС позволило интегрировать кэш второго уровня прямо в микросхему процессора. Такая структура характерна, в частности, для процессора Pentium III версии Coppermine. Его кэш L2 имеет размер 256 Кбайт и восьмиканальную множественно-ассоциативную организацию. Благодаря расположению на одной микросхеме с процессором этот кэш соединен с таковым 256-разрядной шиной.

Читатель наверняка задался вопросом: что же все-таки лучше — интегрировать кэш второго уровня в микросхему процессора или реализовать его вне таковой. Внешний кэш можно сделать большим, но зато он не может соединяться

с процессором такой широкой шиной, как внутренний кэш, поскольку для этого потребуется слишком много выводов и увеличится потребление энергии выходными повторителями. Кроме того, внешние кэши имеют меньшую тактовую частоту. L2-кэш процессора Katmai работает на вдвое меньшей тактовой частоте, чем процессор, тогда как Coppermine — на полной тактовой частое процессора. Размещение L2-кэша на микросхеме процессора уменьшает время ожидания и увеличивает полосу пропускания за счет использования боле широкой шины, в результате чего значительно повышается производительность. Основным недостатком интегрированного кэша второго уровня является увеличение размеров микросхемы процессора, затрудняющее ее производство.

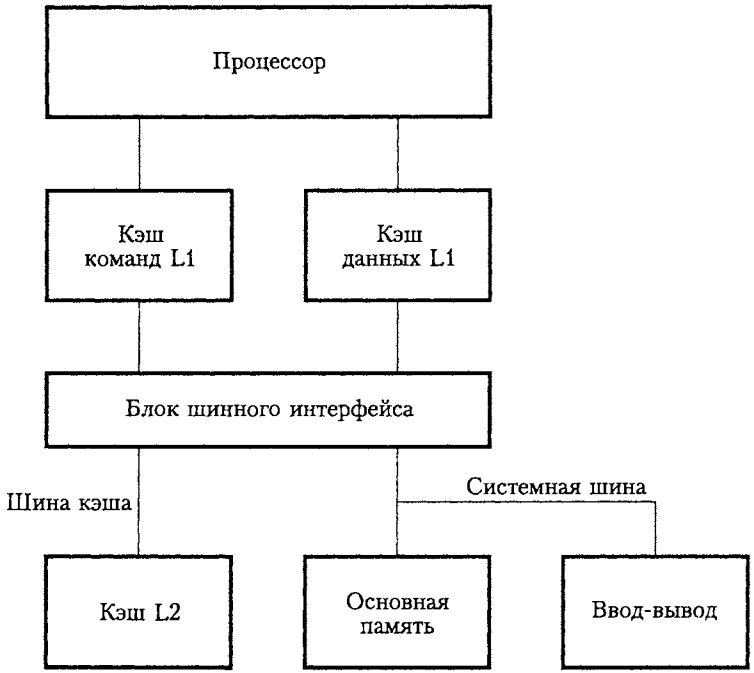


Рис. 5.24. Кэши и внешние соединения процессора Pentium III

Кэши процессора Pentium 4

У процессора Pentium 4 может быть до трех уровней кэш-памяти. Кэш-память первого уровня состоит из отдельных кэша команд и кэша данных. Кэш данных емкостью 8 Кбайт имеет четырехканальную множественно-ассоциативную организацию. Размер блока составляет 64 байта. Для записи данных в кэш применяется протокол сквозной записи. Целочисленные данные извлекаются из кэша за два такта. Микросхемы Pentium 4 могут работать на тактовых частотах свыше 1,3 ГГц, то есть для доступа к данным требуется менее 2 нс. В кэше команд содержатся не обычные машинные команды, а их декодированные версии (подробнее об этом рассказывается в главе 11).

Кэш второго уровня емкостью 256 Кбайт имеет восьмиканальную множественно-ассоциативную организацию. Размер его блока составляет 128 байт. Для записи данных применяется протокол обратной записи. Время доступа составляет семь тактов.

Оба кэша, L1 и L2, реализованы на микросхеме процессора. Архитектура Pentium 4 позволяет добавить в микросхему и кэш L3, однако он используется только в процессорах для серверных систем.

5.6. Производительность

Важнейшими факторами коммерческого успеха компьютера любой марки являются его производительность и стоимость. Цель создателей любой системы, конечно же, состоит в достижении наивысшей производительности при минимальной цене. Как правило, усилия конструкторов направлены на то, чтобы увеличить производительность системы, не повышая ее цену. Мерой оценки результата является *соотношение цены и производительности*. В данном разделе речь пойдет о производительности памяти и средствах ее повышения.

Производительность компьютерной системы зависит от того, насколько быстро команды программы могут извлекаться из памяти в процессор и насколько быстро они могут выполняться. Факторы, от которых зависит скорость выполнения команд, обсуждаются в главах 7 и 8; там же приводятся и дополнительные схемы, применяемые для ускорения этого процесса. А сейчас мы хотим сконцентрировать ваше внимание на подсистеме памяти.

Иерархия памяти, описанная в разделе 5.4, разработана исходя из оптимального соотношения цены и производительности. Суть иерархической организации памяти состоит в том, что процессор должен иметь возможность обращаться к памяти большого объема за минимальное время. Каждому уровню иерархии отводится очень важная роль. При этом особенно важны скорость и эффективность пересылки данных между различными уровнями. Хорошо, если обмен данными с наиболее быстрыми устройствами выполняется на скорости этих устройств. Если обмен производится между быстрым и медленным устройствами, достичь высокой скорости невозможно, но в случае параллельного доступа к более медленным устройствам к ней вполне можно приблизиться. Эффективным способом реализации параллельного доступа является чередование операций.

5.6.1. Чередование операций

Если основная память компьютера состоит из набора физически раздельных модулей с собственными буферными регистрами адреса (ABR) и данных (DBR), возможен одновременный доступ к более чем одному модулю. Таким образом можно повысить общую скорость пересылки слов в основную память и из нее.

Для определения среднего количества модулей, которые могут быть заняты в процессе вычисления, важно знать, как между ними распределены адреса. Существует два метода распределения адресов между модулями. Согласно первому из

них, генерируемые процессором адреса декодируются так, как показано на рис. 5.25, *а*. Старшие k бит определяют один их n модулей, а младшие m бит — конкретное слово этого модуля. При обращении к последовательным адресам, как, скажем, в том случае, когда блок данных пересылается в кэш, в операции участвует только один модуль. В то же самое время устройства, поддерживающие прямой доступ к памяти (ПДП), могут обращаться к информации из других модулей памяти.

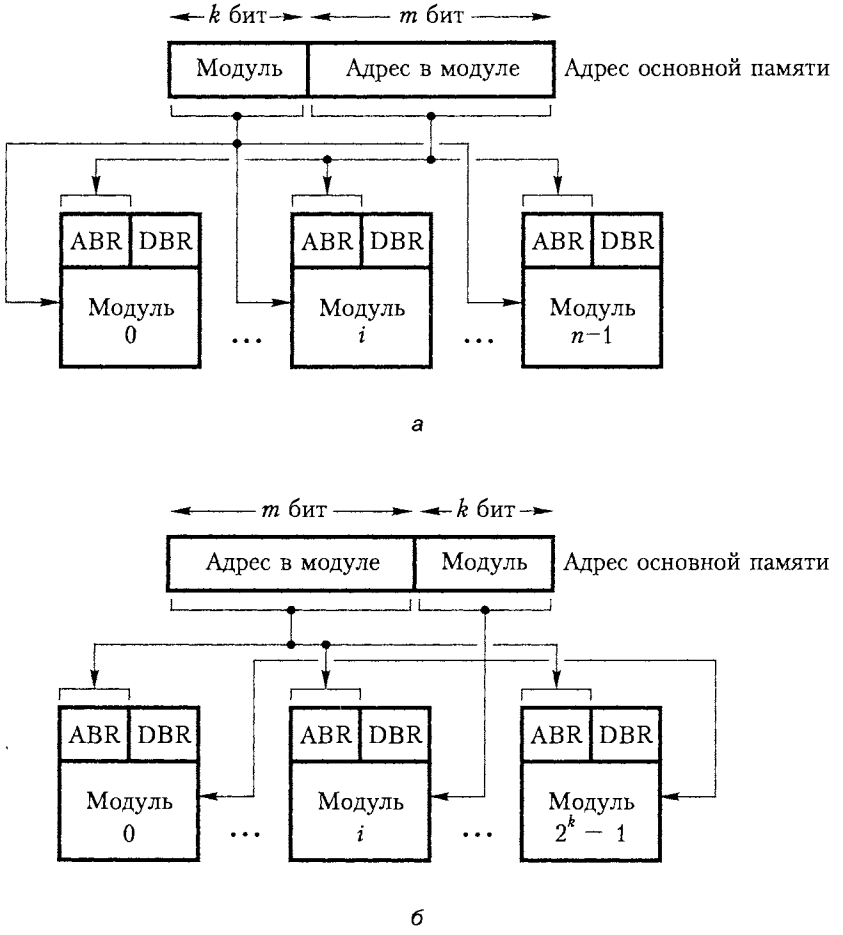


Рис. 5.25. Адресация многомодульной системы памяти: последовательные слова располагаются в одном модуле (*а*), в последовательных модулях (*б*)

Второй и более эффективный способ адресации модулей продемонстрирован на рис. 5.25, *б*. Он называется *чередованием адресов памяти*. Согласно этому методу, младшие k бит адреса памяти выбирают модуль, а старшие m бит — конкретное слово данного модуля. При таком способе адресации последовательные адреса указывают на разные модули. Любой компонент системы, генерирующий запросы на

доступ к последовательным адресам памяти, может поддерживать одновременную активность нескольких модулей. Благодаря этому ускоряется доступ к блоку данных и эффективнее используется вся система памяти. Чередование адресов памяти возможно лишь при условии, что количество модулей равно 2^k; иначе в адресном пространстве памяти будут пробелы.

Пример 5.1

Чередование адресов оказывает существенное влияние на быстродействие памяти. Проанализируем время, уходящее на пересылку блока данных из основной памяти в кэш в случае промаха чтения. Предположим, что у нас имеется кэш с блоками размером восемь слов, как в примере из раздела 5.5. После неудачной операции чтения блок, содержащий требуемое слово, должен быть скопирован из основной памяти в кэш. Кроме того, сделаем предположение, что аппаратное обеспечение имеет следующие характеристики. На пересылку адреса в основную память уходит один такт. Память состоит из относительно медленных микросхем DRAM, так что на доступ к первому слову уходит 8 тактов, а на доступ к каждому последующему — по 4 такта. (В разделе 5.2.3 рассказывалось, что при чтении последовательных элементов данных из одной строки ячеек адрес этой строки декодируется только один раз. После этого поочередно задаются адреса последовательных столбцов массива, и на доступ к соответствующим словам требуется вполовину меньше времени.) Кроме того, один такт необходим для отправки слова в кэш.

В случае использования одного модуля памяти время загрузки блока в кэш вычисляется так:

$$1 + 8 + (7 \times 4) + 1 = 38 \text{ (тактов)}$$

Теперь предположим, что память состоит из четырех модулей с чередующимися адресами (рис. 5.25, *б*). Когда начальный адрес блока достигает памяти, все четыре модуля начинают процедуру доступа к нужным данным с использованием старших битов адреса. После 8 тактов в регистре DBR каждого модуля оказывается одно слово данных. Эти слова по очереди пересылаются в кэш в течение последующих четырех тактов. В течение этого времени в каждом модуле выполняется обращение к следующему слову. Полученные слова, опять-таки, пересылаются в кэш в течение четырех тактов. Таким образом, общее время, необходимое на загрузку блока данных из памяти с чередованием адресов, составляет:

$$1 + 8 + 4 + 4 = 17 \text{ (тактов)}$$

Получается, что чередование более чем вдвое сокращает время пересылки блока.

В разделе 5.2.4 было сказано, что в микросхемах SDRAM технология чередования используется для ускорения доступа к последовательным словам данных. Память в большинстве микросхем SDRAM разделена на два или четыре банка массивов ячеек с чередованием адресов. Это позволяет ускорить передачу блока данных в основную память и из нее.

5.6.2. Частота попаданий и накладные расходы при промахах

Отличным показателем эффективности конкретной реализации иерархии памяти является частота попаданий при доступе к информации на разных уровнях иерархии. Напомним, что попаданием называется успешное обращение к данным в кэше. Отношение количества попаданий к общему количеству попыток доступа называется *частотой попаданий*, а отношение количества промахов к общему количеству попыток доступа — *частотой промахов*.

В идеале вся иерархия памяти должна представляться процессору единым запоминающим устройством, время доступа к которому равно времени доступа к кэш-памяти на микросхеме процессора, а размер равен размеру магнитного диска. Степень приближения к этому идеалу зависит от частоты попаданий на разных уровнях иерархии памяти. В случае высокопроизводительных компьютеров частота попаданий должна превышать 0,9.

Производительность системы очень зависит от того, какие действия выполняются в случае промаха. Дополнительное время, уходящее на копирование необходимой информации в кэш, называется *накладными расходами при промахах*. Эти накладные расходы выливаются в простой процессора, поскольку он не может функционировать без команды или обрабатываемых ею данных. В общем случае накладные расходы — это время, необходимое для пересылки блока данных от самого медленного устройства в иерархии данных к самому быстрому. При использовании эффективных механизмов пересылки данных между различными устройствами общее время задержки значительно сокращается. Одним из таких механизмов является чередование адресов памяти, о котором говорилось в предыдущем разделе.

Пример 5.2

Теперь давайте рассмотрим влияние кэша на общую производительность компьютера. Предположим, частота попаданий равна h, накладные расходы при промахе (то есть время доступа к информации в основной памяти) составляет M (нс), а время доступа к информации в кэше — C (нс). Среднее время доступа, каким оно представляется процессору, таково (параметры системы те же, что в примере 5.1):

$$t_{ave} = hC + (1 - h)M$$

Если у компьютера вообще нет кэша, то при наличии быстрого процессора и типичной DRAM на каждое обращение к памяти для чтения данных уходит 10 тактов. Предположим, что в компьютере имеется кэш с блоками по 8 слов и основная память с чередованием адресов. Тогда, как следует из раздела 5.6.1, на загрузку блока в кэш необходимо 17 тактов. Предположим, что 30 % команд в типичной программе выполняют чтение или запись данных. Это означает, что на каждые 100 выполненных команд приходится 130 обращений к памяти. Предположим, что частота попаданий в кэш составляет 0,95 для команд и 0,9 для данных, а также что накладные расходы при промахах операций чтения и записи одинаковы. Тогда

приблизительная оценка повышения производительности в результате использования кэша будет такой:

$$\frac{\textit{Время без кэша}}{\textit{Время с кэшем}} = \frac{130 \times 10}{100(0{,}95 \times 1 + 0{,}05 \times 17) + 30(0{,}9 \times 1 + 0{,}1 \times 17)} = 5{,}04$$

Получается, что при наличии кэша компьютер работает впятеро быстрее.

Интересно посмотреть, насколько эффективен этот кэш по сравнению с так называемым идеальным кэшем, частота попаданий которого равна 100 % (при наличии такого кэша любое обращение к памяти выполняется за 1 такт). Вот приблизительная оценка относительной производительности двух указанных кэшей:

$$\frac{100(0{,}95 \times 1 + 0{,}05 \times 17) + 30(0{,}9 \times 1 + 0{,}1 \times 17)}{130} = 1{,}98$$

Это означает, что реальный кэш создает для процессора такую среду, где он, процессор, эффективно работает с большой основной памятью на основе DRAM, которая представляется ему лишь в два раза медленнее, чем кэш.

В данном примере мы для упрощения приняли, что для доступа к кэшу на микросхеме процессора и к основной памяти через системную шину применяется одна и та же тактовая частота. Однако высокопроизводительные процессоры работают на частоте, значительно превышающей частоту системной шины — как правило, в десятки раз. Поэтому давайте рассмотрим, как кэш влияет на систему такого типа.

Пример 5.3

Предположим, у нас имеется один кэш, реализованный на микросхеме процессора, и основная память на основе микросхем SDRAM. Мы также предполагаем, что тактовая частота системной шины в два раза меньше тактовой частоты процессора. Как и в примере 5.2, кэш состоит из блоков по 8 слов, и частота попаданий в кэш составляет 0,95 для команд и 0,9 для данных. Временная диаграмма SDRAM такая же, как на рис. 5.9. Единственным ее отличием является то, что пакет состоит не из четырех а из восьми слов данных. Таким образом, в соответствии с рис. 5.9, с момента выдачи сигнала RAS на пересылку блока данных между основной памятью и кэшем уходит 14 тактов. Поскольку сигналы RAS и CAS генерируются контроллером памяти (рис. 5.11), необходим еще один такт, в течение которого процессор должен отослать контроллеру памяти адрес первого слова блока. Следовательно, всего на пересылку блока требуется 15 тактов. Такты, показанные на рис. 5.9, — это такты системной шины. Если тактовая частота процессора вчетверо выше, на пересылку блока из восьми слов между процессором и основной памятью уходит 60 тактов процессора. Обратите внимание, что в схеме, представленной на рис. 5.9, на пересылку одного слова между процессором и основой памятью требуется 9 тактов шины — 8 тактов, показанных на рис. 5.9, плюс один такт, необходимый для пересылки адреса контроллеру памяти. Как видите, на доступ к одному слову основной памяти уходит 36 тактов процессора. И это несмотря на то, что обращение к слову в кэше выполняется за один такт процессора!

Вот что получается, если повторить вычисления из примера 5.2 для других исходных данных:

$$\frac{\textit{Время без кэша}}{\textit{Время с кэшем}} = \frac{130 \times 36}{100(0{,}95 \times 1 + 0{,}05 \times 60) + 30(0{,}9 \times 1 + 0{,}1 \times 60)} = 7{,}77$$

Таким образом, если учесть различие в скорости функционирования процессора и системной шины, кэш дает даже больший выигрыш в производительности.

В предшествующих примерах мы различали частоту попаданий в кэш для данных и для команд. Хотя в обоих случаях она может превышать 0,9, обычно для команд этот показатель выше, чем для данных. В общем случае частота попаданий зависит от организации кэша и от схемы доступа к командам и данным, осуществляемого конкретной программой.

Как же повысить частоту попаданий? Очевидно, это можно сделать путем увеличения объема кэша, но не следует забывать, что при этом возрастет его стоимость. Второй способ заключается в увеличении размера блока при неизменном общем размере кэша, что даст возможность полнее использовать свойство локализации ссылок в пространстве. Если для вычислений необходимы все элементы большого блока, лучше всего загружать их в кэш одним большим массивом, а не по частям. При этом можно будет воспользоваться преимуществами параллельного доступа к данным в памяти с чередованием адресов. Но увеличение размера блока не может быть беспредельным, поскольку начиная с некоторого момента частота попаданий уже не только не повышается, но даже начинает снижаться. Если блок слишком велик, то, пока он находится в кэше, к некоторым его элементам вообще не выполняются обращения. Другими словами, чтобы достичь наивысшей производительности, необходимо подобрать оптимальный размер блока. На практике чаще всего используются блоки размером от 16 до 128 байт.

Напоследок отметим, что способ сокращения накладных расходов промаха все-таки существует. Он заключается в использовании технологии сквозной загрузки, применяемой при считывании в кэш новых блоков команд и данных. Суть ее состоит в следующем: вместо того чтобы ожидать пересылку из памяти всего блока, процессор должен продолжать свою работу, как только в кэше окажется необходимое слово.

5.6.3. Кэши на микросхеме процессора

Когда информация пересылается между разными микросхемами, во входных и выходных вентилях микросхем происходят значительные задержки. Следовательно, самым оптимальным решением, способным увеличить скорость функционирования кэша, является размещение его на микросхеме процессора. Однако пространство на микросхеме процессора предназначается и для многих других функций, поэтому возможный размер кэша здесь очень ограничен.

Микросхемы всех высокопроизводительных процессоров содержат хотя бы небольшой кэш. Некоторые процессоры, и в частности процессоры 68040, Pentium III, Pentium 4, имеют по два отдельных кеша, один для команд, а другой для

данных. К числу тех, которые ограничиваются единым кешем, относится процессор ARM710T.

Комбинированный кэш для команд и данных может обеспечить более высокую частоту попаданий, поскольку он гибче в отношении размещения новых данных, но зато к раздельным кэшам возможен одновременный доступ, что также ведет к повышению производительности. Недостатком раздельных кэшей является сложность управляющих схем.

Обычно высокопроизводительные процессоры имеют два уровня кэш-памяти. Кэш L1 интегрируется в микросхему процессора, а кэш L2 чаще всего бывает внешним, имеет больший объем и реализуется на основе микросхемы SRAM. Случается, что и кэш второго уровня интегрируют в микросхему процессора, как в процессоре Pentium III версии Coppermine, о котором рассказывалось в разделе 5.5.4 (в таком случае его объем несколько меньше, чем у внешнего кэша).

Если процессор имеет два уровня кэш-памяти, доступ к кэшу первого уровня должен выполняться предельно быстро, чтобы не задерживать работу процессора. Правда, доступ к кэшу не может осуществляться так же быстро, как к регистрам, поскольку кэш намного больше и сложнее по своей структуре. Поэтому доступ к кэшу обычно ускоряют путем параллельного доступа к нескольким словам, которые затем по очереди используются процессором. Эта технология применяется во многих современных процессорах.

Кэш второго уровня может быть более медленным, но он должен иметь больший объем, чтобы обеспечить высокую частоту попаданий. Скорость его функционирования имеет сравнительно небольшое значение, поскольку она влияет только на накладные расходы, связанные с промахами кэша L1. Типичная рабочая станция может содержать кэш первого уровня объемом в несколько десятков килобайтов и кэш второго уровня объемом в несколько мегабайтов.

Наличие кэша второго уровня значительно снижает влияние скорости основной памяти на производительность компьютера. Среднее время доступа к памяти, каким оно представляется процессору в системе с двухуровневым кэшем, рассчитывается следующим образом:

$$t_{ave} = h_1C_1 + (1 - h_1)h_2C_2 + (1 - h_1)(1 - h_2)M$$

где:

h_1 — частота попаданий в кэш L1;

h_2 — частота попаданий в кэш L2;

C_1 — время доступа к информации в кэше L1;

C_2 — время доступа к информации в кэше L2;

M — время доступа к информации в основной памяти.

Количество промахов кэша L2, определяемое выражением $(1 - h_1)(1 - h_2)$, должно быть предельно низким. Если значения h_1 и h_2 составляют порядка 90 %, то общее количество промахов, требующих обращения к основной памяти, составит менее 1 %. Таким образом, накладные расходы M с точки зрения производительности будут очень незначительны. В упражнении 5.18 вам предлагается выполнить количественную оценку соответствующих показателей.

5.6.4. Другие способы увеличения быстродействия

Кроме уже рассмотренных нами ключевых конструкторских решений существует еще несколько возможностей повышения производительности системы памяти. О трех из них рассказывается в настоящем разделе.

Буферизация записи

При использовании протокола сквозной записи каждой операции записи соответствует операция сохранения нового значения в основной памяти. Если процессор будет ждать окончания этой операции, его работа очень сильно замедлится. Однако дальнейшая работа процессора в ближайшее время, как правило, не зависит от результата операции записи, так что ему незачем дожидаться ее завершения. А раз так, в систему памяти можно добавить *буфер записи*, предназначенный для временного хранения результатов операций записи. Процессор должен поместить в него запрос на запись и продолжать свою работу. А тем временем, уже без его участия, запрос будет отослан в основную память, когда та не будет занята выполнением запросов на чтение. Запросы на чтение должны обслуживаться немедленно, поскольку процессор обычно не может продолжать работу без тех данных, которые он запросил. Поэтому у запросов на чтение более высокий приоритет, чем у запросов на запись.

В буфере записи может скопиться много запросов. Поэтому не исключены ситуации, когда запросы на чтение ссылаются на данные, которые еще находятся в буфере записи. Для обеспечения корректной работы системы адреса данных, подлежащих чтению из памяти, обязательно сравниваются с адресами в буфере записи, и в случае их совпадения используются данные из буфера записи.

При использовании протокола обратной записи ситуация становится несколько иной. В этом случае в результате операции записи изменяются только данные в кэше. Но что происходит, когда по причине промаха нужно прочитать в кэш новый блок данных, поместив его на место блока с измененными и еще не сохраненными данными? Перед удалением из кэша измененный блок, конечно же, записывается в основную память. Если обратная запись производится часто, процессору приходится подолгу дожидаться выполнения операции чтения из памяти очередного блока данных. Поэтому разумнее сначала прочитать новый блок, чтобы процессор мог продолжить работу, а потом заняться записью в память удаленного из кэша блока. Для этого нужен быстрый буфер записи, куда перед чтением нового блока будет временно помещаться старый. Когда новый блок будет прочитан из основной памяти, в нее можно будет записать содержимое старого блока. Как видите, буфер записи необходим при любом из протоколов записи.

Упреждающая выборка

Рассказывая о технологиях кэширования, мы предполагали, что данные копируются в кэш тогда, когда они нужны процессору. Это означает, что процессор обращается к памяти, и если в кэше не оказывается нужных данных, таковые загружаются из основной памяти. Однако в этом случае процессору приходится ждать поступления новых данных (накладные расходы промаха), что, конечно же, замедляет его работу.

Чтобы процессор не простаивал, необходимые данные следует помещать в кэш еще до того, как они потребуются. Проще всего это сделать программным путем, для чего в системе команд процессора должна быть предусмотрена команда упреждающей выборки. Ее выполнение приводит к загрузке в кэш указанных в ней данных, как в случае промаха. Однако процессор не ждет адресованных данных. Команда упреждающей выборки вставляется в программу для того, чтобы данные были загружены в кэш до того, как они потребуются программе. Предполагается, что выборка производится, пока процессор занят выполнением других команд, не вызывающих промахов чтения, так что доступ к основной памяти осуществляется параллельно с процессом вычислений.

Команды упреждающей выборки могут быть вставлены в программу как программистом, так и компилятором. Конечно, лучше возложить эту задачу на компилятор, тем более что справляется он с такой задачей вполне успешно. С использованием команд упреждающей выборки связаны и некоторые издержки, поскольку эти команды увеличивают длину программы. Более того, некоторые из них могут загружать в кэш данные, не используемые последующими командами. Так может произойти в том случае, если заранее помещенные в кэш данные будут вытеснены другими данными еще до того, как программа успеет к ним обратиться. Однако в целом программная упреждающая выборка положительно сказывается на производительности, и многие процессоры поддерживают эту функцию.

Упреждающая выборка может выполняться и аппаратным путем, но для этого необходимы дополнительные схемы, определяющие последовательность обращений к памяти и прогнозирующие следующие обращения. Существует множество стратегий и соответствующих им схем прогнозирования, но их анализ выходит за рамки нашего издания.

Процессор Intel Pentium 4 поддерживает и программную и аппаратную упреждающую выборку. У него имеются специальные команды, копирующие блок данных в кэш заданного уровня. Средства аппаратной упреждающей выборки копируют данные в кэш второго уровня, используя специальный алгоритм, который позволяет определить, насколько интенсивно они использовались прежде.

Кэш без блокировок

Если команды упреждающей выборки постоянно прерывают нормальное выполнение программы, целесообразность их применения оказывается под большим сомнением. Так бывает, если упреждающая выборка мешает обращениям к кэшу. Получается, что на то время, пока выполняется выборка данных, кэш заблокирован для процессора. Для решения этой проблемы архитектуру кэша можно модифицировать таким образом, чтобы процессор мог обращаться к кэшу параллельно с выборкой из памяти новых данных. Более того, желательно, чтобы кэш поддерживал несколько параллельных операций выборки.

Кэш, в котором обработка промахов и команд упреждающей выборки может откладываться, с тем чтобы процессор мог обратиться к имеющимся в нем данным, называется *кэшем без блокировок* (lockup-free). Поскольку кэш способен обработать за раз лишь один промах, в него должна быть включена схема для отслеживания остальных промахов, ожидающих очереди на обработку. Для этого нужны

специальные регистры, в которых хранилась бы информация об отложенных промахах. Кэши без блокировок впервые появились в 1980-х годах, в компьютерах серии Cyber, производившихся компанией Control Data.

Программная упреждающая выборка производится с целью предотвратить блокировку кэша в случае промахов при выполнении операции чтения. Но существует еще одно, более важное обстоятельство. Как вы понимаете, в процессорах с конвейерной организацией, при которой параллельно выполняются несколько команд, промах чтения одной команды может задержать выполнение остальных. Кэш без блокировок сокращает вероятность таких задержек. Мы еще вернемся к этому вопросу в главе 8, когда будем обсуждать конвейерное выполнение команд.

5.7. Виртуальная память

В большинстве современных компьютерных систем физическая основная память не так велика, как используемое процессором адресное пространство. Например, адресное пространство процессора, генерирующего 32-разрядные адреса, имеет размер 4 Гбайт. Размер основной памяти типичного компьютера варьируется от нескольких сотен мегабайтов до гигабайта. Если программа не помещается в основную память, то те ее части, которые в данный момент не выполняются, хранятся во вторичном запоминающем устройстве, чаще всего на магнитном диске. Безусловно, перед выполнением необходимая часть программы должна быть перемещена в основную память. Если основная память заполнена, новый сегмент программы должен заменить какой-нибудь из старых сегментов. В современных компьютерах перемещение программы и данных между основной памятью и вторичными запоминающими устройствами выполняется операционной системой автоматически. При этом прикладному программисту не нужно беспокоиться об ограничениях, налагаемых доступным объемом основной памяти.

Технологии автоматического перемещения в основную память сегментов программ и данных, потребовавшихся для выполнения программы, называются технологиями управления *виртуальной памятью*. Программы, а в ходе их выполнения и процессор, ссылаются на пространство команд и данных, не зависимое от реального физического пространства основной памяти. Генерируемые процессором двоичные адреса команд и данных называются *виртуальными* или *логическими* адресами. Объединенными усилиями соответствующих аппаратных и программных компонентов они транслируются в реальные физические адреса. Если виртуальный адрес указывает на часть пространства программы или данных, расположенную в физической памяти, доступ к нему выполняется немедленно, но если этот адрес указывает не на основную память, соответствующий сегмент программы или данных сначала должен быть перемещен в основную память.

На рис. 5.26 проиллюстрирована типичная организация виртуальной памяти. Трансляцию виртуальных адресов в физические выполняет специальный аппаратный блок, называемый *модулем управления памятью* или же *диспетчером памяти* (Memory Management Unit, MMU). Когда нужные данные (или команды) отсутствуют в основной памяти, диспетчер перемещает их туда с диска. Для перемещения данных используется механизм ПДП, о котором рассказывалось в главе 4.

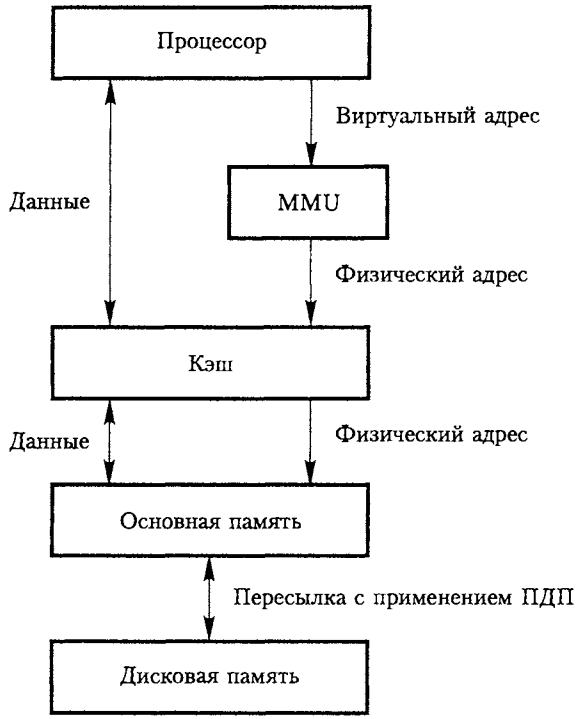


Рис. 5.26. Организация виртуальной памяти

5.7.1. Преобразование адресов

Простейший метод преобразования виртуальных адресов в физические основывается на предположении, что все программы и данные состоят из сегментов фиксированной длины, называемых *страницами*, которые, в свою очередь, состоят из блоков слов, последовательно расположенных в памяти. Размер страницы обычно варьируется от 2 до 16 Кбайт. Страница является базовой единицей информации, перемещаемой между основной памятью и диском по требованию механизма преобразования адресов. Страницы не должны быть слишком маленькими, поскольку время доступа к магнитному диску (составляет несколько миллисекунд) намного больше времени доступа к основной памяти. Значительная часть этого времени уходит на поиск данных на диске. Найденные данные пересылаются со скоростью несколько мегабайт в секунду. С другой стороны, если страница слишком велика, большая ее часть, скорее всего, не будет использована, но место в основной памяти она, конечно же, будет занимать

Все это напоминает концепции кэш-памяти, рассмотренные в разделе 5.5. Кэш сглаживает разницу в быстродействии процессора и основной памяти, а механизм управления виртуальной памятью делает то же самое в отношении основной памяти и вторичного запоминающего устройства. Концептуально технологии управления виртуальной памятью и кэшем очень близки, а их различия связаны главным образом со спецификой реализации.

Итак, метод преобразования адресов основывается на концепции страниц фиксированной длины, схематически представленной на рис. 5.27. Каждый сгенерированный процессором виртуальный адрес, будь то адрес для операции выборки команды или для чтения и записи операнда, интерпретируется как *номер виртуальной страницы* (старшие разряды) и *смещение* (младшие разряды) байта или слова от начала страницы. Информация о местонахождении каждой страницы в основной памяти содержится в *таблице страниц*. Она включает адрес основной памяти, по которому хранится страница, и данные о ее текущем состоянии. Область основной памяти, где может находиться одна страница, называется *страничным блоком*. Начальный адрес таблицы страниц хранится в *базовом регистре таблицы страниц*. Добавив номер виртуальной страницы к содержимому этого регистра, вы получите адрес нужного элемента таблицы страниц. А в самом этом элементе хранится начальный адрес страницы, если, конечно, она имеется в основной памяти.

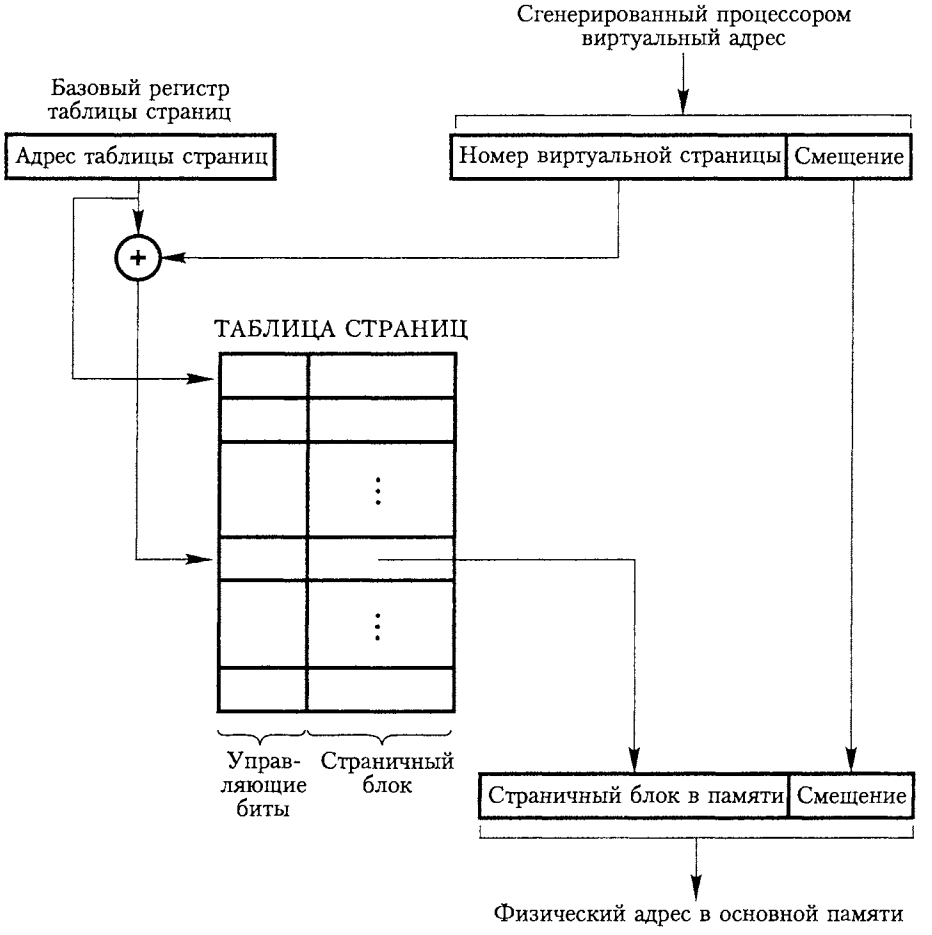


Рис. 5.27. Преобразование адресов виртуальной памяти

Кроме адреса страницы каждый элемент таблицы страниц содержит несколько управляющих битов, которые определяют состояние страницы, находящейся в основной памяти, и еще один бит, указывающий, хранится ли страница в памяти. Последний бит позволяет операционной системе пометить страницу как отсутствующую в памяти, не удаляя ее на самом деле. Еще один бит указывает, была ли страница модифицирована за то время, пока она находилась в основной памяти. Как и в случае кэш-памяти, на основании этой информации принимается решение о том, записывать ли страницу снова на диск перед ее удалением из основной памяти (когда нужно освободить место для другой страницы). Остальные управляющие биты действуют в соответствии с различными ограничениями, налагаемыми на доступ к странице. Например, программе могут быть предоставлены полные права на чтение и запись страницы или же только на ее чтение.

Диспетчер памяти, то есть MMU, использует информацию из таблицы страниц для выполнения каждой операции чтения или записи. Поэтому было бы целесообразно хранить эту таблицу прямо в нем. Но, к сожалению, она слишком велика, а блок управления памятью обычно интегрирован в микросхему процессора (вместе с кэшем первого уровня), куда невозможно добавить такой большой фрагмент памяти. Поэтому таблица страниц содержится в основной памяти. Диспетчер памяти может хранить небольшую ее часть, которая включает элементы, соответствующие недавно использовавшимся страницам. Практически это маленький кеш, обычно называемый *буфером быстрого преобразования адреса* (Translation Lookaside Buffer, TLB). У него то же назначение и такой же принцип действия, как и у любой другой кэш-памяти. Кроме элемента таблицы страниц в TLB должен содержаться виртуальный адрес этого элемента. На рис. 5.28 показан один из вариантов организация TLB на основе ассоциативного отображения. Существуют и TLB с множественно-ассоциативной организацией.

Исключительно важно, чтобы содержимое TLB соответствовало содержимому таблицы страниц в памяти. Когда операционная система изменяет содержимое таблицы страницы, она должна одновременно пометить соответствующие элементы TLB как недостоверные. Для этого в каждом таком элементе имеется специальный управляющий бит. Если элемент помечается как недостоверный, он обновляется в ходе обычной операции, выполняемой MMU, когда нужных данных в TLB не оказывается.

Преобразование адресов осуществляется следующим образом. Получив виртуальный адрес, MMU ищет в TLB заданную страницу. Если нужная запись находится в TLB, из нее тут же извлекается физический адрес страницы. В случае промаха, то есть отсутствия записи в TLB, она считывается из таблицы страниц в основной памяти, и TLB обновляется.

Когда программа генерирует запрос на доступ к странице, отсутствующей в основной памяти, происходит *ошибка страницы*. В этом случае перед продолжением операции вся страница должна быть перемещена с диска в основную память. Для этого MMU обращается к операционной системе, генерируя исключение (прерывание). В результате выполнение активной задачи прерывается, а управление передается операционной системе. Операционная система копирует запрошенную страницу с диска в основную память и возвращает управление прерванной

задаче. Поскольку на пересылку страницы уходит довольно много времени, операционная система может приостановить выполнение задачи, вызвавшей ошибку страницы, и активизировать другую задачу, страницы которой имеются в основной памяти.

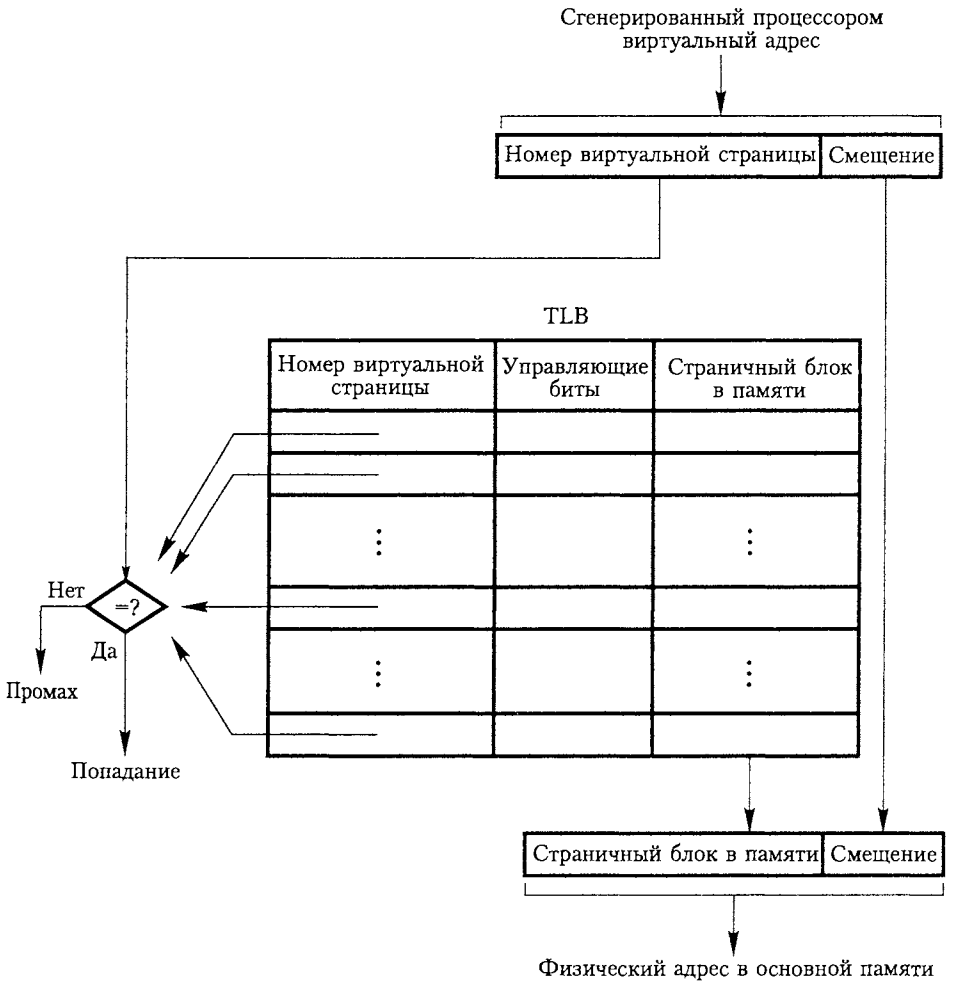


Рис. 5.28. Принцип действия TLB с ассоциативным отображением

Важно, чтобы после приостановки задачи ее выполнение было продолжено. Поскольку ошибка страницы происходит при обращении некоторой команды к операнду, отсутствующему в основной памяти, прерывание производится до завершения выполнения этой команды. Когда выполнение задачи возобновляется, выполнение команды нужно продолжить либо с той точки, где оно было прервано, либо начать сначала. Выбор определяется архитектурой конкретного процессора.

Если с диска нужно переместить новую страницу, а память уже заполнена, приходится удалять одну из страниц, уже имеющихся в памяти. Правильный выбор удаляемой страницы так же важен, как правильный выбор заменяемого блока в кэше, и здесь, конечно же, учитывается тот факт, что большую часть времени выполнение программы ограничивается несколькими локализированными областями. Поскольку основная память значительно больше кэш-памяти, в ней можно держать очень большие фрагменты программы. Поэтому частота обмена информацией с диском может быть сравнительно невысокой. К операции замены страниц применимы концепции, подобные применяемым в алгоритме LRU, а индикаторами использования страниц могут служить управляющие биты в таблице страниц. В одном из простейших алгоритмов замены задействован единственный управляющий бит, устанавливаемый при обращении к соответствующей странице в 1. Время от времени операционная система очищает этот бит во всех записях таблицы страниц, отмечая таким образом все эти страницы как давно не использовавшиеся.

Перед удалением из основной памяти модифицированная страница опять должна быть записана на диск. Однако протокол сквозной записи, который может успешно использоваться в кэш-памяти, для виртуальной памяти совершенно не подходит. Время доступа к диску настолько велико, что нет смысла часто обращаться к нему при необходимости записать небольшой объем данных.

Процесс преобразования адресов в диспетчере памяти сам требует некоторого времени, основная часть которого уходит на поиск записей в TLB. Поскольку принцип локализации ссылок действует и в этом случае, велика вероятность того, что в ряде последовательных преобразований будет использоваться адрес одной и той же страницы. Особенно часто так бывает при выборке команд. Поэтому среднее время преобразования адресов можно сократить, добавив в процессор один или несколько специальных регистров для хранения номера виртуальной страницы и адреса физического страничного блока, применявшихся в последнем преобразовании. Доступ к информации из этих регистров будет осуществляться даже быстрее, чем доступ к TLB.

5.8. Требования к управлению памятью

Рассказывая о концепциях виртуальной памяти, мы предполагали, что в системе выполняется только одна большая программа. Если вся программа не помещается в физическую память, ее части (страницы) перемещаются с диска в основную память, когда приходит время их выполнения. И хотя мы упоминали о том, что управление перемещением сегментов программы между основной памятью и диском производится соответствующим программным обеспечением, деталей этого процесса мы не касались.

Программы управления виртуальной памятью являются частью операционной системы компьютера. Программы операционной системы удобно объединять в виртуальное адресное пространство, которое называется *системным пространством* и отделяется от виртуального адресного пространства, где выполняются прикладные программы. Последнее называется *пространством пользователя*. Фактически

пользовательских адресных пространств может быть множество, по одному для каждого пользователя. В этом случае для каждой прикладной программы создается отдельная таблица страниц. MMU считывает ее адрес из базового регистра таблицы страниц. При переключении от одной программы к другой операционная система изменяет содержимое этого регистра. Таким образом, физическая основная память делится между активными страницами системного пространства и нескольких пользовательских пространств. Причем в каждый конкретный момент доступны страницы только одного из этих пространств.

В любой компьютерной системе, в которой в основной памяти сосуществуют независимые пользовательские программы, используется тот или иной вид *защиты*. Ни одна программа не должна иметь возможности разрушать данные или команды других программ. Такая защита может обеспечиваться несколькими способами. Мы же для начала рассмотрим наиболее простую ее форму. Напомним, что в простейшей системе процессор может находиться в одном из двух состояний — *супервизора* или *пользователя*. При выполнении подпрограмм операционной системы процессор работает в режиме супервизора, а при выполнении прикладных программ — в режиме пользователя. В последнем случае запрещено выполнение некоторых машинных команд. *Привилегированные команды*, выполняющие такие операции, как, скажем, модификация базового регистра таблицы страниц, могут быть реализованы только в режиме супервизора. Это значит, что пользовательская программа не имеет доступа ни к таблице страниц, ни к другим адресным пространствам.

Иногда возникает необходимость предоставить прикладной программе доступ к страницам, принадлежащим другой прикладной программе. Для этого операционная система может включить эти страницы в оба адресных пространства, поместив ссылки на них в две разные таблицы страниц. Для управления правами доступа, предоставляемыми каждой из программ, могут быть применены соответствующие управляющие биты в таблице страниц. Например, одной программе может быть позволено и считывать и записывать некоторую страницу, а другой — только считывать.

5.9. Внешние запоминающие устройства

Обсуждавшаяся в предыдущих разделах полупроводниковая память удовлетворяет далеко не всем потребностям компьютера, связанным с хранением данных. Основным ее недостатком является высокая стоимость хранения единицы информации. Большинству компьютерных систем нужны запоминающие устройства очень большой емкости, роль которых выполняют магнитные диски, оптические диски и магнитные ленты, обычно называемые *вторичными* или *внешними* запоминающими устройствами.

5.9.1. Жесткие магнитные диски

Как следует из названия, магнитный диск состоит из одного или нескольких дисков, нанизанных на один шпиндель. Диск имеет тонкое магнитное покрытие, как

правило, наносимое на обе его стороны. Вся эта конструкция с постоянной скоростью вращается с помощью мотора, так что намагниченные поверхности движутся над головками, выполняющими операции считывания и записи (рис. 5.29, *а*). Головка представляет собой электромагнит, состоящий из сердечника с обмоткой, по которой пропускается ток (рис. 5.29, *б*).

Информация записывается на магнитный слой с помощью электромагнитных импульсов нужной полярности, генерируемых током в обмотке. Головка намагничивает находящуюся непосредственно под ней поверхность диска, изменяя вектор намагниченности ее частиц. Она же используется и для чтения информации. В этом случае изменение магнитного поля вблизи головки, вызываемое движением диска, индуцирует напряжение в обмотке, которая в данном режиме играет роль сенсора. Управляющие схемы фиксируют полярность этого напряжения и определяют состояние магнитного покрытия диска. При чтении данных фиксируются только изменения магнитного поля под головкой. Если двоичные значения 0 и 1 представлены двумя противоположными состояниями намагниченности, то напряжение в головке индуцируется только при переходе от 0 к 1 или от 1 к 0. Однако если на диске хранится строка нулей или единиц, напряжение индуцируется только в начале и конце этой строки. Для определения количества последовательно расположенных нулей или единиц необходима синхронизирующая информация, позволяющая отсчитать количество позиций, которые имеют одинаковую намагниченность. В некоторых ранних конструкциях дисков синхронизирующая информация записывалась на отдельную дорожку, где намагниченность изменялась для каждого последовательного бита. Используя данные этой дорожки, управляющие схемы диска могли правильно считывать информацию с других дорожек.

Современный подход заключается в комбинировании синхронизирующей информации с данными. Разработано несколько разных схем такого кодирования. Одна из них, показанная на рис. 5.29, *в*, известна как *фазовое* или *манчестерское кодирование*. Согласно этой схеме, намагниченность изменяется для каждого бита данных. Обратите внимание, что такое изменение происходит в середине той области дорожки, которая предназначена для записи одного бита. Недостатком манчестерской технологии является малая плотность хранения битов. Для представления каждого бита необходимо пространство, позволяющее дважды изменить намагниченность. Манчестерская технология является очень наглядным примером самосинхронизации, но на практике используются более компактные схемы, обеспечивающие большую плотность записи данных. Их обсуждение выходит за рамки нашей книги.

Для достижения максимальной плотности битов и предельной надежности чтения и записи головки чтения/записи должны располагаться очень близко к вращающимся поверхностям дисков. Когда диски вращаются с постоянной скоростью, между ними и головками создается поток воздуха, отталкивающий головки от поверхности. Для противодействия этой силе головки монтируются на пружинной основе, прижимающей их к поверхности. Гибкое пружинящее соединение головки с местом ее крепления позволяет ей парить над поверхностью на нужном расстоянии, несмотря на возможные небольшие изменения уровня поверхности диска, который может не быть идеально плоским.

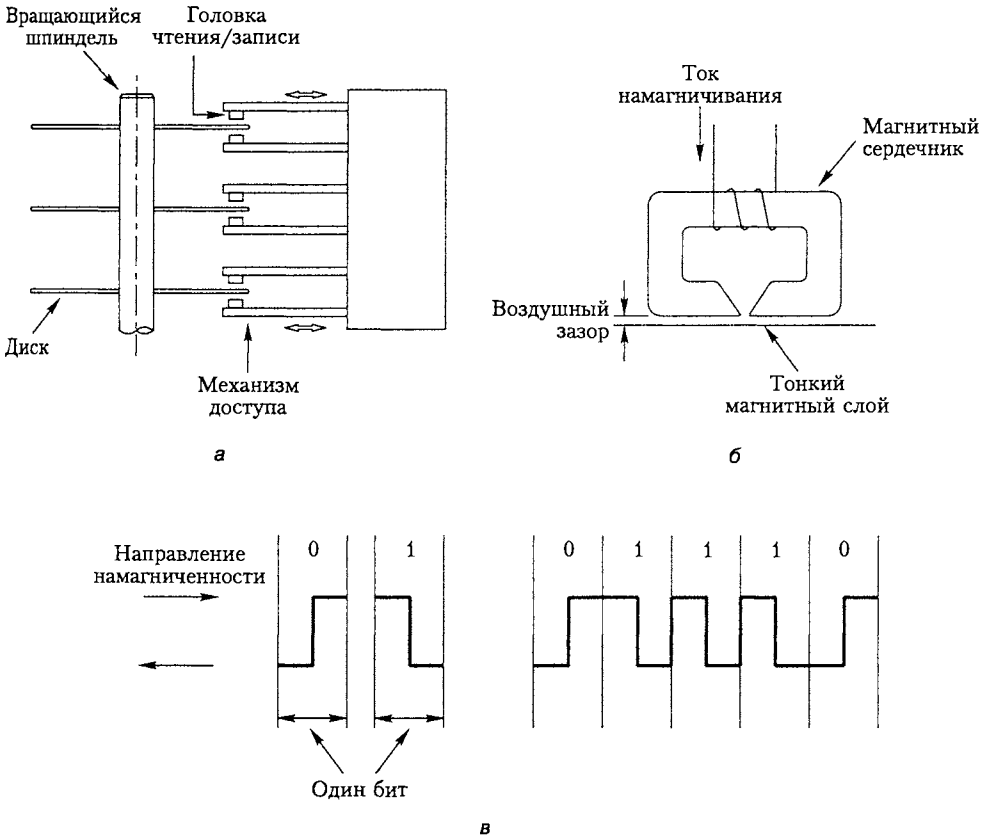


Рис. 5.29. Магнитный диск: механическое устройство (*а*); головка чтения/записи (*б*); представление бита при фазовом кодировании (*в*)

В большинстве современных дисковых устройств диски и головки чтения/записи помещены в плотно закрытый корпус, воздух в котором фильтруется. Такая технология изготовления жестких дисков называется *винчестерской*. Она позволяет располагать головки чтения/записи ближе к поверхности дисков, поскольку в закрытом корпусе отсутствуют частицы пыли, представляющие большую проблему для открытых конструкций. А чем ближе головки расположены к поверхности, тем ближе друг к другу можно расположить биты на дорожке, и тем ближе друг к другу можно расположить сами дорожки. Таким образом, винчестерские диски, обычно называемые просто винчестерами, по сравнению с открытыми устройствами при одинаковом физическом размере имеют значительно большую емкость. Кроме того, магнитная запоминающая среда в них защищена от загрязнения.

Головки чтения/записи жесткого диска подвижны. Каждой поверхности соответствует одна головка. Все вместе они объединены в гребнеобразную конструкцию, перемещающуюся по радиусу между центром и краем диска, обеспечивая доступ к разным дорожкам, как показано на рис. 5.29, *а*. Для чтения и записи заданной дорожки вся система головок должна быть перемещена к этой дорожке.

Дисковую систему можно разбить на три основные составляющие: первая — это набор дисковых пластин, которые, вместе взятые, составляют собственно *диск*; вторая, *дисковый привод* или *дисковод*, представляет собой электромеханическое устройство, которое вращает диск и перемещает головки; третья объединяет все электронные схемы, управляющие работой системы и называемые *контроллером диска*. Контроллер может быть реализован в виде отдельного модуля или заключен внутрь корпуса дисковой системы. Следует отметить, что термин *диск* часто применяется для обозначения объединенного устройства, состоящего из диска и дисковода. В таком значении использовать его будем и мы, но лишь в тех случаях, когда это не будет приводить к неоднозначности.

Организация данных и доступ к данным на диске

Как данные организованы на диске, показано на рис. 5.30. Все поверхности диска разбиты на концентрические *дорожки*, а каждая дорожка, в свою очередь, делится на *секторы*. Набор расположенных друг над другом дорожек на всех поверхностях диска образует логический *цилиндр*. Доступ к данным всех дорожек цилиндра осуществляется без перемещения головок чтения/записи. Для доступа к данным задается номер поверхности, номер дорожки и номер сектора. Операции Read и Write начинаются на границах секторов.

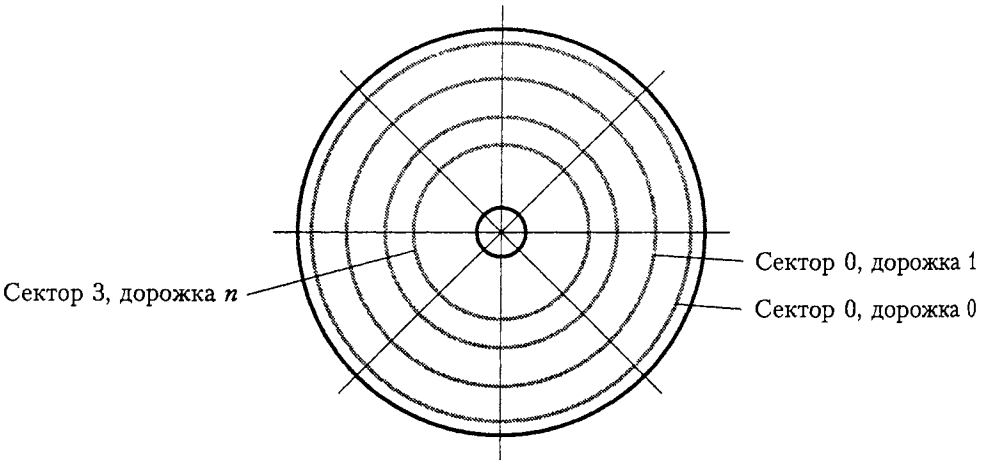


Рис. 5.30. Организация одной поверхности диска

На каждой дорожке биты данных хранятся последовательно. Сектор обычно содержит 512 байт данных, но он может быть и другого размера. Данные предваряются *заголовком сектора*, содержащим идентификационную (адресную) информацию, которая используется для поиска нужного сектора на выбранной дорожке. За данными следует ряд дополнительных битов, которые составляют *код коррекции ошибок* (Error Correction Code, ECC). Биты ECC используются для выявления и исправления ошибок, которые могут произойти при чтении или записи 512 байт данных. Для того чтобы секторы было легче разграничить, между ними оставляется небольшое пространство — межсекторный интервал.

Дорожки неформатированного диска не содержат никакой информации. В ходе форматирования диск физически разбивается на дорожки и секторы. При этом могут быть выявлены дефектные секторы и даже целые дорожки. Дисковый контроллер запоминает информацию о таких дефектных участках и исключает их. Таким образом, объективным показателем вместимости данного диска является объем памяти на диске после форматирования. Информация о форматировании составляет около 15 % от всей хранящейся на диске информации. Она состоит из заголовков секторов, битов ECC и межсекторных интервалов. В типичном компьютере диск разбит еще и на логические разделы. На диске всегда имеется один первичный раздел, но может быть еще и несколько дополнительных.

Судя по рис. 5.30, все дорожки диска разбиты на одинаковое количество секторов. Следовательно, все они имеют одинаковую емкость. Причем на внутренних дорожках информация располагается более плотно, чем на внешних. Такая организация данных характерна для многих дисков, поскольку она очень упрощает электронные схемы, управляющие доступом к данным. Емкость диска можно увеличить, расположив на внешних, более длинных дорожках больше секторов, чем на внутренних. Правда, управляющие схемы в этом случае придется сделать более сложными. Такая организация обычно применяется в больших дисках.

Время доступа

Время задержки между получением адреса данных и началом их пересылки условно делится на две составляющие. Первая из них, называемая *временем поиска*, — это время, необходимое для перемещения головок чтения/записи к нужной дорожке. Оно зависит от исходной позиции головок относительно заданной дорожки. В среднем время поиска составляет от 5 до 8 мс. Вторая составляющая — это *задержка позиционирования*, то есть время, которое проходит от момента подведения головки к нужной дорожке до того момента, когда под головкой окажется начало нужного сектора. В среднем оно составляет половину времени одного оборота диска. Сумма двух указанных составляющих задержки называется *временем доступа*. Если в одной операции пересылается всего несколько секторов данных, время доступа обычно на порядок больше времени пересылки данных.

Стандартные магнитные диски

Наиболее широко используемые в настоящее время 3,5-дюймовые (по диаметру) высокоскоростные диски имеют следующие характеристики:

- 15000 дорожек на каждой из 20 поверхностей;
- в среднем 400 секторов на дорожке;
- 512 байт данных в секторе.

Таким образом, емкость форматированного диска равна $20 \times 15000 \times 400 \times 512 \approx \approx 60 \times 10^9 \approx 60$ Гбайт. Среднее время поиска составляет 6 мс. Диск вращается со скоростью 10 000 об/мин, так что средняя задержка позиционирования равна 3 мс (половина от времени одного оборота). Среднее время внутренней пересылки данных от дорожки в буфер данных дискового контроллера составляет 34 Мбайт/с. Если диск подключен к шине SCSI, его внешняя скорость пересылки

может быть равной 160 Мбайт/с. Таким образом, для сглаживания различия в скоростях пересылки данных необходима схема буферизации, описанная в следующем разделе.

Существуют диски и меньших габаритов. Например, на 1-дюймовом диске может храниться 1 Гбайт данных. Такой диск размером со спичечный коробок весит менее унции (28,3 г). Его можно использовать в портативном оборудовании и карманных устройствах. В цифровой камере на нем может храниться порядка 1000 фотографий. Интересно, что первый диск емкостью 1 Гбайт был создан IBM в 1980 году. Этот диск имел размер кухонного оборудования, весил 250 кг и стоил 40 тыс. долларов.

Буфер данных

С остальной частью компьютерной системы жесткий диск соединяется так же, как любые другие устройства. Обычно для этого применяется стандартная шина, в частности шина SCSI, описанная в разделе 4.7.2. Дисковое устройство с интегрированным интерфейсом SCSI обычно называют SCSI-диском. Шина SCSI позволяет пересылать данные с гораздо более высокой скоростью, чем они могут считываться с дисковых дорожек. Поэтому для сглаживания разницы в скорости в дисковое устройство интегрируется *буфер данных*, представляющий собой полупроводниковую память, в которой может храниться несколько мегабайтов данных. Запрошенные данные пересылаются между дисковыми дорожками и буфером со скоростью, определяемой скоростью вращения диска. А пересылка данных между этим буфером и другими подключенными к шине устройствами может выполняться с максимальной скоростью шины.

У буфера данных имеется еще одно назначение: он часто используется для кэширования данных диска. Когда диск получает запрос на чтение, контроллер проверяет, нет ли нужных данных в кэше (буфере). Если они там присутствуют, поместить их на шину SCSI можно не в течение нескольких миллисекунд, как в случае непосредственного обращения к диску, а за несколько микросекунд. В противном случае данные считываются с диска обычным образом и сохраняются в кэше. Поскольку велика вероятность того, что в последующих обращениях к диску будут запрошены данные, расположенные в непосредственной близости от текущих данных, контроллер может инициировать чтение в кэш большего количества данных, потенциально сократив время ответа на следующий запрос. Обычно кэш достаточно велик и в нем хранятся целые дорожки, так что пересылка содержимого дорожки в буфер данных начинается сразу после того, как только головка чтения/записи оказывается над этой дорожкой.

Контроллер диска

За функционирование дискового устройства отвечает *дисковый контроллер*, который обеспечивает интерфейс между этим дисковым устройством и шиной, соединяющей его с остальной частью компьютерной системы. Контроллер может использоваться для управления более чем одним дисковым устройством (рис. 5.31).

Рис. 5.31. Диски, соединенные с системной шиной

Контроллер диска соединяется непосредственно с процессорной системной шиной или шиной расширения, например с PCI. Он содержит множество регистров, содержимое которых может считываться и записываться операционной системой. Таким образом, операционная система взаимодействует с контролером диска точно так же, как с другими интерфейсами ввода-вывода (глава 4). Для пересылки данных между диском и основной памятью контроллер использует механизм прямого доступа к памяти. Речь идет о пересылке данных в буфер и из буфера, входящего в состав модуля дискового контролера. Операционная система инициирует передачу с помощью запросов считывания и записи с последующей загрузкой в регистры контроллера необходимой адресной и управляющей информации. Речь идет в первую очередь о следующей информации:

- адресе основной памяти — адресе первого блока или слова, подлежащего пересылке или предназначенного для приема данных;
- адресе на диске — местоположении сектора, содержащего начало нужного блока слов;
- количестве слов в пересылаемом блоке (счетчике слов).

Адрес на диске — это логический адрес, генерируемый операционной системой. Соответствующий физический адрес на диске может быть иным. Например, если при форматировании диска будут обнаружены дефектные секторы, дисковый контроллер подставит вместо них другие. Обычно на каждой дорожке специально для этой цели имеется несколько запасных секторов, а в цилиндре может быть даже запасная дорожка. Перечислим основные функции контроллера.

- Поиск — контроллер заставляет дисковод переместить головку чтения/записи к нужной дорожке.
- Чтение — контроллер инициирует операцию чтения начиная с адреса, заданного в адресном регистре диска. Считанные с диска данные собираются в слова и помещаются в буфер данных для пересылки в основную память. Количество слов задается в регистре счетчика слов.

- Запись — контроллер пересылает данные на диск с использованием управляющего метода, сходного с методом выполнения операции чтения.
- Контроль ошибок — контроллер формирует для прочитанных из сектора данных код корректировки ошибок (ECC) и сравнивает его со значением ECC, прочитанным с диска. Если эти два значения не совпадают, он пытается исправить ошибку. В противном случае он генерирует прерывание, чтобы проинформировать операционную систему о произошедшей ошибке. В ходе операции записи контроллер вычисляет значение ECC для помещаемых в сектор данных и сохраняет его на диске.

Если диск соединяется с шиной, поддерживающей пакетную пересылку данных, контроллер может использовать ее преимущества. Например, контроллер SCSI-диска применяет протокол шины SCSI, описанной в главе 4.

Программное обеспечение и операционная система

Все операции пересылки данных, в которых участвуют дисковые устройства, инициируются операционной системой. Диск является энергонезависимым запоминающим устройством, так что на нем хранится и сама операционная система. В процессе работы компьютера модули операционной системы по мере необходимости загружаются в основную память.

Когда питание отключается, содержимое основной памяти теряется. После того как питание включается снова, операционная система копируется в основную память в ходе процесса, называемого *загрузкой*. Для того чтобы система могла инициировать загрузку, небольшая часть основной памяти реализуется в виде энергонезависимой ROM. В ней хранится маленькая *программа-монитор*, которая может считать и записать данные из основной памяти, а также прочитать с диска один блок данных, хранящийся по адресу 0. Этот блок, называемый *блоком начальной загрузки*, содержит программу-загрузчик. Сначала ROM-монитор загружает в основную память блок начальной загрузки, а затем загрузчик из этого блока загружает с диска основные части операционной системы.

Доступ к диску выполняется значительно медленнее, чем доступ к основной памяти — главным образом из-за задержки поиска. После того как операционная система инициирует операцию обращения к диску, она обычно передает управление какой-нибудь другой задаче, чтобы максимально рационально использовать время до завершения передачи данных. Когда данные будут переданы, контроллер диска сообщит об этом операционной системе посредством прерывания.

Если в компьютерной системе имеется много дисков, операционной системе может потребоваться прочитать или записать данные на несколько из них. Для того чтобы наиболее эффективно выполнить эту операцию, можно совместить пересылку, управляемую механизмом прямого доступа к памяти, с поиском на другом диске. Управление такими параллельными операциями ввода-вывода выполняет операционная система.

Гибкие диски

Рассмотренные выше устройства называются жесткими дисками. Существуют также *гибкие диски* — более простые и дешевые съемные диски меньшего размера, представляющие гибкие пластиковые *дискеты* с магнитным покрытием. Такая

дискета заключена в пластиковый конверт с небольшим отверстием, через которое с ней контактирует головка чтения/записи. В центре дискеты имеется еще одно отверстие, необходимое для ее вращения.

Одной из простейших схем, используемых в первых гибких дисках для записи данных, было описанное выше фазовое или манчестерское кодирование. Диски с закодированной таким образом информацией назывались дисками *одинарной плотности*. В современных гибких дисках применяется более сложный вариант этой же схемы, называемый *двойной плотностью*. Она вдвое увеличивает плотность записи информации, хотя и усложняет схему дискового контроллера.

Основными достоинствами гибких дисков являются их малая стоимость и то, что их легко вынимать из компьютера и использовать в качестве переносных носителей. Но, с другой стороны, гибкие диски отличаются очень малой емкостью, большим временем доступа и к тому же они менее надежны, чем жесткие диски. Стандартная дискета имеет диаметр 3,25 дюйма и может содержать 1,44 или 2 Мбайт данных. Существуют гибкие диски и большого объема. Одна из их разновидностей, так называемые *ZIP-диски*, позволяет хранить свыше 100 Мбайт данных. Правда, с появлением перезаписываемых компакт-дисков, о которых рассказывается ниже, популярность технологии гибких дисков несколько уменьшилась.

Дисковые массивы RAID

За последнее десятилетие скорость процессоров невероятно возросла и продолжает удваиваться каждые 18 месяцев. Скорость полупроводниковой памяти увеличивается не так быстро. И очень сильно отстают от них по данному показателю жесткие диски, время доступа которых по-прежнему измеряется в миллисекундах. Они, конечно же, тоже постоянно усовершенствуются, но это усовершенствование выражается главным образом в увеличении емкости.

Высокопроизводительные устройства, как правило, очень дороги. К счастью, достичь высокой производительности за приемлемую цену иногда можно и путем использования нескольких недорогих устройств, работающих в параллельном режиме. В главе 12 будет рассказано, как организуется параллельная работа нескольких процессоров в мультипроцессорных системах. Аналогичным образом для создания высокопроизводительного запоминающего устройства можно задействовать набор магнитных дисков.

В 1988 году специалистами из Калифорнийского университета Беркли была предложена система хранения на основе нескольких дисков. Ее назвали RAID (Redundant Array of Inexpensive Disks — избыточный массив недорогих дисков). Одновременное использование нескольких дисков позволяет не только увеличить объем и быстродействие запоминающей системы, но и повысить ее надежность. Для RAID-массивов было разработано шесть разных конфигураций, отвечающих различным задачам. Они названы уровнями RAID, хотя никакой иерархии не представляют.

RAID 0 — это базовая конфигурация дискового массива, предназначенная для повышения производительности системы. Один большой файл разбивается на несколько частей, которые записываются на разные диски. Это называется *расслоением данных*. При обращении к такому файлу для чтения диски могут передавать данные параллельно, так что общее время его пересылки по сравнению со

временем хранения на одном диске уменьшается во столько раз, сколько дисков в RAID-массиве. Однако время доступа к конкретному диску, то есть задержки на поиск и позиционирование, не уменьшается. А поскольку все диски работают независимо друг от друга, они имеют разное время доступа. Следовательно, необходимо произвести буферизацию получаемых от дисков фрагментов файлов, в ходе которой файл собирается из частей и отправляется процессору как единое целое. Это простейший способ функционирования дискового массива, при котором уменьшается только время пересылки данных.

Архитектура RAID 1 позволяет повысить надежность хранения данных путем записи их идентичных копий не на одном, а на двух дисках. Такие диски называются *зеркальными*. Если один из них выходит из строя, операции чтения и записи продолжаются с зеркальным диском. Это довольно дорогой способ повышения надежности хранения, поскольку он требует дублирования всех дисков системы.

Уровни RAID 2, RAID 3 и RAID 4 предназначены для повышения надежности системы с помощью различных схем контроля четности, не требующих полного дублирования дисков. Вся информация, предназначенная для контроля четности, хранится на диске.

В RAID 5 также используется схема выявления ошибок, основанная на контроле четности. Однако информация, предназначенная для контроля четности, распределяется между всеми дисками.

Разработано и несколько гибридных схем. Так, RAID 10 представляет собой дисковый массив, одновременно выполняющий функции RAID 0 и RAID 1.

Концепция RAID получила коммерческое признание. В частности, компания Dell Computer Corporation предлагает свои продукты на основе RAID 0, RAID 1, RAID 5 и RAID 10. Цены на магнитные носители за последние несколько лет значительно снизились, так что теперь уже неуместно говорить как о «недорогих» лишь о дисках, составляющих RAID-массив. Поэтому термин RAID переопределен как массив «независимых (independent) дисков».

Типы дисковых устройств

Большая часть дисковых устройств была разработана для стандартных шин. Производительность дискового устройства зависит от его внутренней структуры и интерфейса, связывающего его с остальной частью системы. Его стоимость определяется прежде всего емкостью, а также объемом продаж конкретного продукта.

Диски ATA/EIDE. Наиболее широкое распространение в мире компьютеров получили всем известные персональные компьютеры IBM PC, впервые выпущенные компанией IBM в 1980 году. Для их шины IBM PC bus был разработан дисковый интерфейс. Современная расширенная версия этого интерфейса, ставшая всеобщим стандартом, называется EIDE (Enhanced Integrated Drive Electronics — усовершенствованные электронные схемы управления встроенным дисководом) или ATA (Advanced Technology Attachment — технологически усовершенствованное подключение). Многие производители выпускают большой диапазон дисков с интерфейсом EIDE/ATA. Такие диски могут подключаться прямо к шине PCI (описана в разделе 4.7.1), используемой во многих ПК. Наборы микросхем материнской платы для процессора Intel Pentium включают контролер, позволяющий

подключать диски EIDE/ATA прямо к материнской плате. Важным преимуществом дисков EIDE/ATA является их низкая стоимость, а основным недостатком — то, что при параллельном использовании двух дисков для повышения производительности каждому из них нужен отдельный контроллер.

Диски SCSI. Как вы уже знаете, многие диски имеют интерфейс для подключения к стандартной шине SCSI. Обычно такие диски дороже, но их производительность выше, поскольку шина SCSI эффективнее, чем PCI. В частности, шина SCSI поддерживает параллельный доступ к нескольким дисковым устройствам, поскольку интерфейс диска активно подключается к шине SCSI лишь тогда, когда диск готов к пересылке данных. Это особенно полезно в системах, выполняющих огромное количество обращений к файлам небольших размеров, как, например, в компьютерах, используемых в качестве файл-серверов.

Диски RAID. Диски RAID имеют высокую производительность и обеспечивают надежное хранение больших объемов данных. Они применяются в первую очередь в высокопроизводительных системах, а также в системах с повышенной степенью надежности. Однако по мере снижения цен диски RAID все чаще используются в компьютерных системах среднего размера.

5.9.2. Оптические диски

Большие запоминающие устройства можно создавать и на основе оптической технологии. Хорошо знакомые вам компакт-диски (CD), используемые в аудиосистемах, стали первым практическим результатом применения этой технологии. Вскоре после их появления указанная технология была принята в компьютерной среде, для которой разработаны высокоемкие, доступные только для чтения носители, получившие название CD-ROM.

CD первого поколения были разработаны в середине 1980-х годов компаниями Sony и Phillips, опубликовавшими полную спецификацию этих устройств. Технология производства CD основана на возможности использовать цифровое представление аналоговых звуковых сигналов. Для обеспечения высококачественной записи и воспроизведения звука на диск записываются частотные характеристики отсчетов звукового сигнала с частотой дискретизации 44,1 кГц. Такая частота дискретизации вдвое выше частоты исходного звукового сигнала, что позволяет воспроизводить его с большой точностью. Сейчас продолжительность проигрывания записанной на CD музыки должна составлять не менее часа. Время проигрывания CD первых версий не превышало 75 мин, что соответствует количеству информации, равному 3×10^9 бит, то есть 3 Гбит. С тех пор были разработаны устройства и большей емкости. На видео-CD может храниться целый фильм, а для этого, как вы понимаете, требуется на порядок большая емкость, чем у аудио-CD. Мультимедийные CD подходят и для хранения больших объемов компьютерных данных.

Технология CD

Используемая в CD-системах оптическая технология основана на применении лазерного луча. Лазерный луч направляется на поверхность вращающегося диска, вдоль дорожек которого располагаются впадины, отражающие сфокусированный

луч в направлении фотодетектора, фиксирующего записанные на диске двоичные данные. Лазер излучает когерентный свет, состоящий из синхронизированных волн одинаковой длины. Если объединить два одинаковых луча в одной фазе, получится более яркий луч, а если сдвинуть лучи на полфазы, они погасят друг друга. Но если два таких луча будут направлены на фотодетектор, то в первом случае он зафиксирует яркое пятно, а во втором — темное.

Рассмотрим сечение CD (рис. 5.32, *а*). Его нижний слой выполняется из поликарбонатного пластика, играющего роль прозрачной основы. Данные наносятся на поверхность диска в виде *впадин* (pit), чередующихся с плоскими участками — *площадками* (land). Поверх диска с записанной на него информацией наносится тонкий слой отражающего алюминия, а на него — защитное акриловое покрытие. Сверху клеится этикетка. Общая толщина диска составляет 1,2 мм, причем большая ее часть приходится на поликарбонатный пластик, так как остальные слои очень тонкие.

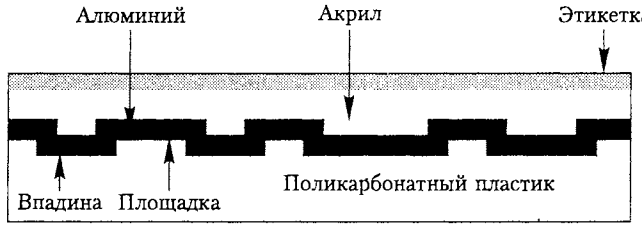
Источник лазерного излучения и фотодетектор располагаются под поликарбонатным пластиком. Лазерный луч скользит по пластику, отражается от алюминиевого слоя и попадает на фотодетектор. Со стороны лазера впадины выглядят как выпуклости по отношению к плоской поверхности.

Что происходит, когда скользящий вдоль диска лазерный луч перемещается от впадины к площадке, видно из рис. 5.32. При вращении диска возможны три разных позиции источника луча и детектора относительно впадин и площадок на его поверхности. Когда свет отражается только от впадины или только от площадки, детектор фиксирует яркое пятно. Однако на границе впадины, глубина которой равна четверти длины волны лазера, ситуация меняется. Волна, отраженная от впадины, смещается на 180° по фазе относительно волны, отраженной от площадки, и они гасят друг друга. Таким образом, на границах впадина—площадка и площадка—впадина детектор не видит отраженного луча и фиксирует темное пятно.

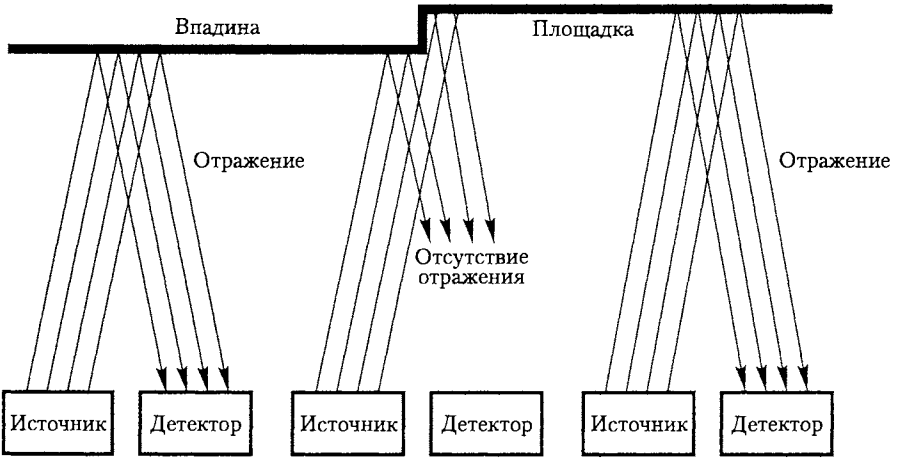
На рис. 5.32, показано несколько переходов между площадками и впадинами. Если каждый переход, фиксируемый как темное пятно, обозначается двоичным значением 1, а поверхность впадины или площадки — двоичным значением 0, результирующая двоичная последовательность будет такой, как на этом рисунке. Она не является непосредственным представлением хранящихся на диске данных. Для CD применяется сложная система кодирования информации. Каждый байт представлен 14-разрядным кодом, позволяющим выявлять и исправлять ошибки, но мы не будем обсуждать его подробно.

Впадины располагаются вдоль дорожек диска. Строго говоря, на диске имеется только одна физическая дорожка, раскручивающаяся по спирали от центра диска к его внешнему краю. Каждый из ее 360-градусных фрагментов рассматривается как отдельная дорожка — по аналогии с магнитными дисками. Диаметр CD равен 120 мм, а посередине диска имеется 15-миллиметровое отверстие. Данные хранятся на дорожках, которые находятся на расстоянии 25–58 мм от центра диска. Расстояние между дорожками составляет 1,6 мк. Впадины имеют ширину 0,5 мк и длину от 0,8 до 3 мк. Всего на диске располагается более 15000 дорожек. (Если бы мы могли раскрутить спиральную дорожку, она вытянулась бы на 5 км!)

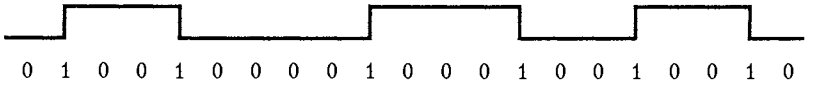
Таким образом, на сантиметр радиуса диска приходится около 6000 дорожек, что гораздо больше максимальной плотности, характерной для магнитных дисков. Для магнитных дисков этот показатель равен 800–2000, а для дискет менее 40–2000 дорожек на сантиметр.



а



б



в

Рис. 5.32. Оптический диск: срез (*а*); переход от впадины к площадке (*б*); хранящаяся на диске двоичная последовательность (*в*)

CD-ROM

Поскольку информация на компакт-дисках хранится в двоичном формате, они могут использоваться в качестве носителей данных в компьютерных системах. Наибольшей проблемой в этом случае становится обеспечение целостности данных. Так как впадины очень малы, процесс их записи достаточно сложен. В аудио- и видеоприложениях незначительное количество ошибок вполне допустимо, поскольку оно не отражается на воспроизведении звука или изображения. А вот в компьютерных приложениях подобных ошибок следует избегать. Но поскольку этого добиться невозможно, необходимы дополнительные биты для выявления и исправления ошибок. Компакт-диск с такими битами называется *CD-ROM*, поскольку после записи производителем его содержимое можно только считывать, как из полупроводниковой ROM.

Данные на дорожках CD-ROM организованы в блоки, которые называются *секторами*. Существует несколько разных форматов секторов. В соответствии с одним из них, Mode 1, размер сектора должен быть равным 2352 байтам. Каждый сектор снабжен 16-байтовым заголовком, который содержит поле синхронизации, используемое для определения начала сектора, и адресную информацию, предназначенную для идентификации сектора. Далее следуют 2048 байт данных. В конце сектора располагаются еще 288 байт, предназначенных для коррекции ошибок. Дорожки имеют переменное количество секторов — чем ближе к краю диска располагается дорожка, тем больше на ней секторов.

Выявление и исправление ошибок производится на нескольких уровнях. Как уже упоминалось в разделе, посвященном описанию компакт-дисков, каждый байт данных кодируется с помощью 14-битового кода, содержащего информацию, на основании которой выполняется устранение ошибок. Этот код позволяет исправлять таковые лишь в одном бите. Ошибки нескольких битов выявляются и устраняются с помощью контрольных битов в конце сектора.

Дисководы CD-ROM вращают диски с разной скоростью. Базовая же скорость вращения, которая обозначается как 1X, составляет 75 секторов в секунду. При использовании формата Mode 1 она соответствует скорости пересылки данных 153600 байт/с (150 Кбайт/с). При такой скорости и таком формате CD-ROM на основе стандартного CD, предназначенного для хранения аудиофайла с временем воспроизведения 75 мин, вмещает около 650 Мбайт данных. Обратите внимание, что скорость дисковода влияет только на скорость пересылки данных, но не на вместимость диска. Скорость CD-ROM определяется относительно базовой скорости — 150 Кбайт/с. Так, современный сорокаскоростной CD-ROM пересылает данные в 40 раз быстрее, чем односкоростной. Но даже эта скорость, составляющая порядка 6 Мбайт/с, значительно ниже скорости пересылки данных магнитным диском, измеряемой десятками мегабайтов в секунду. Еще одной важной характеристикой производительности CD-ROM является время доступа, равное нескольким сотням миллисекунд. Поэтому, с точки зрения производительности CD-ROM значительно уступают магнитным дискам. Их привлекательность определяется малым физическим размером, невысокой стоимостью и тем, что они представляют собой съемные и легко транспортируемые носители достаточно большого объема.

CD-ROM можно поставить в один ряд с такими недорогими портативными носителями, как гибкие диски и магнитные ленты, по сравнению с которыми они, правда, имеют больший объем, а также меньшее время доступа. Благодаря этому CD-ROM широко используются для распространения программного обеспечения, баз данных, больших объемов текстов (книг и библиотек), прикладных программ и видеоигр.

CD-R

Описанные выше компакт-диски доступны только для чтения, а их запись выполняется с помощью специальной процедуры лишь один раз. Сначала с применением мощного лазера изготавливается мастер-диск, на котором выжигаются отверстия в тех местах, где должны располагаться впадины. Затем на его основе создается матрица с выступами на местах впадин. В эту матрицу заливается поликарбонатный пластик, в результате чего получается диск, в точности повторяющий рельеф мастер-диска. Совершенно очевидно, что данный процесс может быть использован только для массового производства компакт-дисков.

В конце 1990-х был разработан новый тип CD, на который данные легко записываются пользователем компьютера. Он был назван *записываемым CD* (CD-Recordable, CD-R). Спиральная дорожка наносится на такой диск в процессе производства, а лазер используется для выжигания отверстий в покрываемом диск слое органического вещества. Когда такое вещество нагревается до критической температуры, оно темнеет. При чтении диска темные пятная отражают меньше света. Произвести запись поверх однажды помещенных на CD-R данных невозможно. Но неиспользованные части диска позднее могут пригодиться для записи дополнительных данных.

CD-RW

Существуют CD-ROM, которые могут многократно перезаписываться пользователем. Они называются *перезаписываемыми CD* (CD-ReWritable, CD-RW).

Базовая структура CD-RW подобна структуре CD-R. Однако вместо органического записываемого слоя здесь используется сплав серебра, индия, сурьмы и теллура. При нагревании и охлаждении этот сплав ведет себя очень интересным образом. Если нагреть его до температуры плавления (500 °C), а затем охладить, он перейдет в аморфное состояние и приобретет способность поглощать свет. Но если указанный сплав будет нагрет до 200 °C и некоторое время выдержан в таком состоянии, то произойдет *отжиг*, в результате чего сплав перейдет в кристаллическое состояние и начнет пропускать свет. Сплав в кристаллическом состоянии представляет площадки диска, а сплав в аморфном состоянии — впадины. Записанные на диск данные можно стереть путем отжига, возвращающего сплав в кристаллическое состояние. Поверх записываемого слоя наносится материал, отражающий свет при чтении диска.

В дисководах CD-RW используется лазерный луч, который имеет три уровня мощности. Самая высокая мощность необходима для нанесения впадин. Мощность средней величины используется для перевода сплава в кристаллическое состояние; она называется «мощностью стирания». А для чтения записанной на диск информации применяется лазер с наименьшей мощностью. Информацию на

диске CD-RW нельзя перезаписывать бесконечное количество раз — современные диски выдерживают до 1000 перезаписей.

Дисководы CD-RW можно применять для работы с другими компакт-дисками, а именно с компакт-дисками для чтения CD-ROM, а также для чтения и записи CD-R. К компьютеру они подключаются через стандартные интерфейсы, в том числе через EIDE, SCSI и USB.

Диски CD-RW представляют собой недорогие носители, подходящие для архивирования больших объемов информации, например баз данных или коллекций фотографических изображений. В настоящее время скорость дисководов CD-RW вполне достаточна для архивирования данных с жестких дисков. Они уверенно вытесняют из обихода дисководы CD-R, поскольку стоят лишь немногим больше таковых, но обладают многими преимуществами, и в частности возможностью многократной перезаписи, что говорит уже само за себя.

Технология DVD

Успех технологии CD и возрастающие требования к вместимости дисков привели к разработке новой технологии, получившей название DVD (Digital Versatile Disk — универсальный цифровой диск). Первый стандарт DVD был разработан в 1996 году консорциумом компаний. Его задачей было хранение полнометражного фильма на одной стороне диска.

Физический размер DVD-диска такой же, как у CD: толщина — 1,2 мм, диаметр — 120 мм. А вот его вместимость благодаря перечисленным ниже изменениям в конструкции значительно выше, чем у CD.

- Вместо инфракрасного лазера с длинной волны 780 мм, который характерен для CD-технологии, в DVD используется красный лазер с длинной волны 635 мм.
- Впадины имеют меньший размер — их минимальная длина равна 0,4 мк.
- Дорожки располагаются ближе друг к другу — расстояние между ними составляет 0,74 мк.

Указанные изменения привели к тому, что DVD способны содержать порядка 4,7 Гбайт данных.

Еще большее увеличение емкости достигается за счет использования двухслойных и двусторонних дисков. Однослойные и односторонние диски, определяемые стандартом DVD-5, имеют практически ту же структуру, что и CD, представленный на рис. 5.32, *а*. В двухслойном диске дорожки наносятся на два слоя, расположенные один поверх другого. Первый слой образует прозрачную основу, как на CD. Но площадки и впадины этого слоя покрываются не отражающим луч алюминием, а полупрозрачным материалом, действующим подобно полуотражающему слою. На его поверхность сначала наносятся данные в виде впадин и площадок, а затем, сверху, — отражающий материал. Для чтения диска лазерный луч фокусируется на нужном уровне. Когда он попадает на первый слой, некоторое количество света, достаточное для того, чтобы детектор зафиксировал впадины и площадки, отражается от полупрозрачного материала. При фокусировке на втором слое он отражается от отражающего материала. В обоих случаях тот слой,

на котором луч не фокусируется, тоже отражает свет, но в таком незначительном количестве, что детектор интерпретирует его как шум. Общая емкость обоих слоев составляет 8,5 Гбайт. Описанная технология определена как стандарт DVD-9.

Два односторонних диска могут быть объединены в структуру, напоминающую сэндвич, так как верхний диск перевернут отражающей стороной вверх. При этом, согласно стандарту DVD-10, могут использоваться два однослойных диска общей емкостью 9,4 Гбайт. Можно объединить и два двухслойных диска общей емкостью 17 Гбайт — такая структура определяется стандартом DVD-18.

Время доступа при использовании дисков DVD примерно то же, что и в случае применения обычных CD. Однако при вращении с той же скоростью данные считываются быстрее благодаря более высокой плотности их размещения.

DVD-RAM

Разработана и перезаписываемая версия DVD-дисков, названная DVD-RAM. При очень большой емкости их недостатком является высокая цена и относительно низкая скорость записи. Для обеспечения корректной записи данных на диск используется процесс, называемый проверкой записи. Его выполняет дисковод DVD-RAM, который считывает содержимое диска и сверяет его с исходными данными.

5.9.3. Накопители на магнитных лентах

Магнитные ленты удобно использовать для хранения больших объемов данных. Чаще всего они применяются для резервного копирования информации с жестких дисков. Принципы хранения информации на ленте те же, что и на магнитном диске (правда, в первом случае магнитный слой наносится на очень тонкую пластиковую ленту шириной 0,5 или 0,25 дюйма). Соответствующие одному символу 7 или 9 бит записываются поперек ленты, перпендикулярно направлению перемотки. Каждой битовой позиции соответствует отдельная головка чтения/записи, так что все биты считываются и записываются параллельно. Один бит символа предназначается для контроля четности.

Данные на ленте организованы в виде записей, разделенных пробелами (рис. 5.33). Движение ленты прекращается только тогда, когда под головкой чтения/записи оказывается пробел. Он достаточно велик, чтобы перед началом следующей записи устройство перемотки могло вернуться к исходной скорости. Если для записи данных на ленту применяется система кодирования, подобная проиллюстрированной на рис. 5.29, *в*, пробелы между записями идентифицируются как области, намагниченность в которых не меняется. Это позволяет отличать пробелы от данных. Для того чтобы помочь пользователю в организации большого количества данных, записи объединяются в группы, называемые *файлами*. Начало файла идентифицируется *меткой файла* (рис. 5.33). Это специальная запись, состоящая из одного или нескольких символов, которой обычно предшествует пробел, более длинный, чем пробелы между записями. Первая запись за меткой файла может использоваться как *заголовок* или *идентификатор* файла. Это дает возможность пользователю находить файлы на ленте, содержащей большое количество файлов.

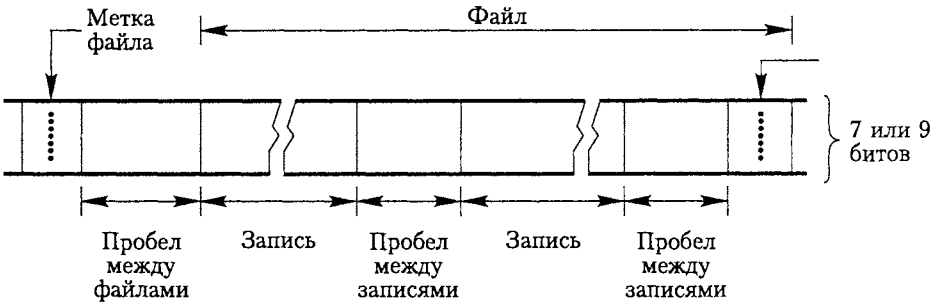


Рис. 5.33. Организация данных на магнитной ленте

Контроллер накопителя на магнитной ленте наряду с чтением и записью позволяет выполнять множество других управляющих команд:

- перемотку ленты;
- перемотку и выгрузку ленты;
- стирание информации с ленты;
- запись метки ленты;
- перемотку ленты на одну запись вперед;
- перемотку ленты на одну запись назад;
- перемотку ленты на один файл вперед;
- перемотку ленты на один файл назад.

Метка ленты отличается от метки файла лишь тем, что используется для идентификации начала ленты. Конец ленты иногда идентифицируется символом EOT (End Of Type), о чем более подробно рассказывается в приложении Д.

Существует два метода форматирования и использования ленты. Согласно первому из них, записи имеют переменную длину. Это позволяет эффективнее использовать ленту, но препятствует обновлению отдельных записей. В соответствии со вторым методом, записи имеют фиксированную длину и их можно обновлять. Хотя на первый взгляд это может показаться значительным преимуществом, но на самом деле не имеет особого значения. Поскольку ленты чаще всего используются для резервного копирования магнитных дисков и архивирования данных, они редко перезаписываются по частям. Обычно лента записывается от начала до конца, и размер записей не играет никакой роли.

Картриджи с магнитными лентами

Системы хранения на магнитных лентах разрабатывались с целью резервного копирования информации, содержащейся на дисковых устройствах. В одной из таких систем, получившей довольно широкое распространение, используются 8-миллиметровые ленты видеоформата, заключенные внутрь кассеты. Такая кассета называется *картриджем*. Емкость картриджа составляет от 2 до 5 Гбайт, а скорость считывания с него данных — несколько сотен килобайтов в секунду. Чтение и запись выполняются системой спиральной развертки, подобной той, что применяется в видеокассетах. Плотность записи данных составляет десятки

миллионов битов на квадратный дюйм. Существуют системы, позволяющие автоматизировать загрузку и выгрузку картриджей таким образом, чтобы десятки гигабайтов данных можно было скопировать с диска без вмешательства оператора.

5.10. Резюме

Память является одним из основных компонентов любого компьютера. Ее емкость и быстродействие в значительной степени определяют производительность всей компьютерной системы. В этой главе мы рассмотрели наиболее важные технологии создания и детали организации памяти и запоминающих устройств.

В результате постоянно проводимых разработок в области полупроводниковой технологии скорость функционирования и емкость микросхем памяти увеличиваются просто с впечатляющей скоростью, а их стоимость в пересчете на бит хранимой информации непрерывно снижается. Однако процессорные микросхемы совершенствуются еще быстрее, и в отношении быстродействия они значительно опередили микросхемы памяти. Для того чтобы в полной мере использовать возможности современных процессоров, компьютер должен обладать большой и быстрой памятью. А поскольку не менее важным показателем является ее стоимость, нельзя просто реализовать всю память на быстродействующих микросхемах SRAM. Поэтому, как было показано в этой главе, проблема решается путем создания иерархии памяти.

На сегодняшний день память достаточно большого объема и с приемлемой стоимостью реализуется на основе микросхем DRAM. Правда, работает она на порядок медленнее быстрого процессора, поэтому для сокращения времени доступа процессора к памяти используется кэш-память на основе микросхем SRAM. Время ожидания памяти является одним из важнейших параметров производительности компьютера, на уменьшение которого постоянно направляются усилия разработчиков. Множество исследований проводится с целью создания схем, позволяющих минимизировать влияние задержки при обращении к памяти. В этой главе было показано, как буферизация записи и упреждающая выборка могут сократить влияние такой задержки путем обращений к памяти в те промежутки времени, когда к ней не производится высокоприоритетный доступ с целью обработки промахов чтения. Время доступа к памяти можно сократить еще одним способом — путем параллельного доступа к последовательным словам. В современные микросхемы памяти закладывается и такая возможность.

Вторичные запоминающие устройства в виде магнитных и оптических дисков располагаются на нижнем уровне иерархии памяти, имеющем наибольшую емкость. Механизм виртуальной памяти делает взаимодействие между диском и основной памятью прозрачным для пользователя. Аппаратная поддержка виртуальной памяти давно уже стала стандартной функцией процессоров.

История развития магнитных дисков служит одним из наиболее впечатляющих примеров эволюции компьютерных технологий. Они всегда были самой медленной частью иерархии памяти. Время от времени, при появлении какой-либо из новых многообещающих технологий, судьба магнитных дисков оказывалась

под вопросом. В начале 1980-х казалось, что в ближайшем будущем их вытеснит технология цилиндрических магнитных доменов. Еще недавно их конкурентами считались флэш-диски и оптические диски. Но магнитные диски не только не вытеснены из обихода, а следовательно, с рынка, а, напротив, сохраняют огромную популярность и постоянно совершенствуются. Увеличивается их емкость, уменьшаются габариты, неуклонно снижается стоимость в пересчете на бит.

Упражнения

5.1. Приведите блок-схему организации модуля памяти 512 К × 8, подобную схеме памяти 8 М × 32, представленной на рис. 5.10.

5.2. Обратимся к рис. 5.6, на котором показана структура ячейки динамической памяти. Предположим, что $C = 50$ фемтофарад (10^{-15} Ф), а утечка тока через транзистор составляет около 9 пикоампер (10^{-12} А). Напряжение на полностью заряженном конденсаторе равно 4,5 В. Ячейка должна быть регенерирована до того, как напряжение упадет ниже 3 В. Оцените минимальную частоту регенерации.

5.3. На рис. 5.8 в нижнем правом углу показаны регистры входных и выходных данных. Приведите схему, реализующую по одному биту каждого из этих регистров, и покажите связи между блоком «Схемы и защелки чтения/записи» и шиной данных.

5.4. Рассмотрим основную память, которая состоит из микросхем SDRAM и имеет такие же временные характеристики, как и микросхема, показанная на рис. 5.9, с той лишь разницей, что в нашем примере длина пакета равна 8. Предположим, что 32 бита данных пересылаются по шине параллельно. Сколько времени при тактовой частоте 133 МГц займет пересылка:

а) 32 байт данных;

б) 64 байт данных.

Каково в каждом из этих случаев время ожидания?

5.5. Опровергните следующее утверждение: «При использовании более быстрого процессора производительность компьютера возрастает пропорционально его быстродействию, даже если быстродействие основной памяти остается прежним.»

5.6. Программа состоит из двух вложенных циклов: меньшего внутреннего и значительно большего внешнего. Ее общая структура приведена на рис. У5.1. Десятичные адреса в памяти показывают местоположение двух циклов, а также начало и конец всей программы. Все фрагменты программы, строки 17–22, 23–164, 165–239 и т. д., содержат команды, которые должны выполняться последовательно. Программа запускается на компьютере, имеющем кэш команд с прямым отображением (рис. 5.15) и такие характеристики:

Размер основной памяти	64 К слов
Размер кэша	1 К слов
Размер блока	128 слов

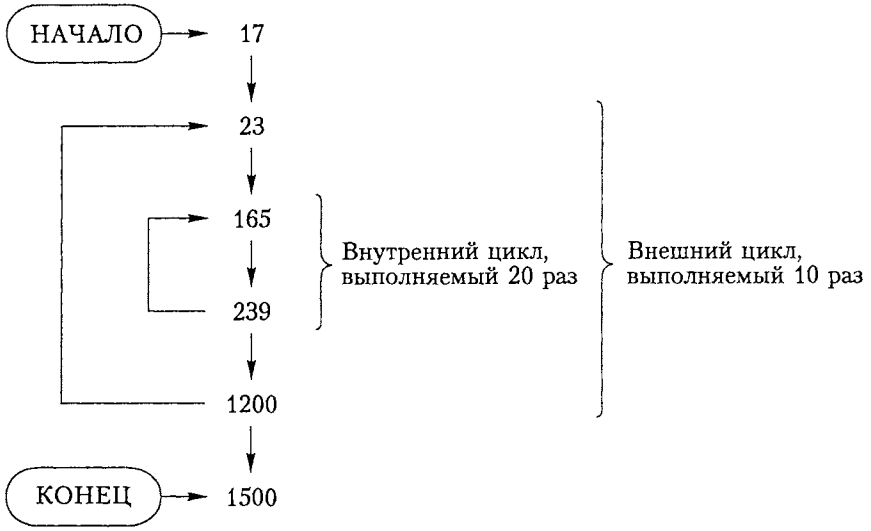


Рис. У5.1. Структура программы для упражнения 5.6

Длительность цикла основной памяти составляет 10 τ, а длительность цикла кэша — τ.

а) Укажите количество битов в полях TAG, BLOCK и WORD адреса основной памяти.

б) Вычислите общее время, уходящее на выборку команд в процессе выполнения программы, приведенной на рис. У5.1.

5.7. В компьютере используется небольшой кэш с прямым отображением, расположенный между основной памятью и процессором. Он вмещает четыре 16-разрядных слова, и с каждым словом ассоциируется 13-разрядный тег, как на рис. У5.2, *а*. Когда в процессе операции чтения происходит промах, запрошенное слово считывается из основной памяти и отсылается процессору. Одновременно оно копируется в кэш, а номер его блока запоминается в соответствующем теге. Рассмотрим следующий цикл в программе, все команды и операнды которой имеют длину 16 бит:

```
LOOP    Add         (R1)+,R0
        Decrement   R2
        BNE         LOOP
```

Предположим, что перед началом этого цикла в регистрах R0, R1 и R2 содержались значения 0, 054E и 3 соответственно. Также предположим, что в основной памяти хранились данные, приведенные на рис. У5.2, *б*, где все они представлены в шестнадцатеричном формате. Цикл начинается по адресу LOOP = 02EC.

а) Каким будет содержимое кэша в конце каждого шага цикла?

б) Допустим, что время доступа к основной памяти составляет 10 τ, а время доступа к кэшу — τ. Вычислите время выполнения каждого шага цикла без учета процессорного времени между циклами памяти.

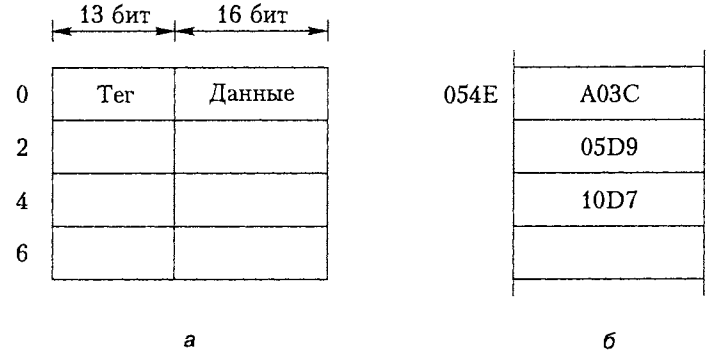


Рис. У5.2. Содержимое кэша и основной памяти в упражнении 5.7: кэш (*а*); основная память (*б*)

5.8. Повторите упражнение 5.7 при условии, что в кэше хранятся только команды. Данные-операнды извлекаются прямо из основной памяти и не копируются в кэш. Почему при этом программа выполняется быстрее, чем при записи в кэш и команд и данных?

5.9. Множественно-ассоциативный кэш состоит из 64 блоков, разделенных на 4-блочные множества. В основной памяти содержится 4096 блоков, каждый из которых включает 128 слов.

а) Какую длину имеет адрес основной памяти?

б) Какую длину имеют его поля TAG, SET и WORD?

5.10. Основная память компьютерной системы содержит 1 M 16-разрядных слов. Кроме того, в системе имеется множественно-ассоциативный кэш с 4 блоками в множестве и 64 словами в блоке.

а) Вычислите количество битов в полях TAG, SET и WORD адреса основной памяти.

б) Предположим, что первоначально кэш пуст. Процессор последовательно выбирает 4352 слова по адресам 0, 1, 2, ..., 4351. Затем он повторяет эту же последовательность выборки еще 9 раз. Оцените коэффициент ускорения за счет использования кэша при условии, что он работает в 10 раз быстрее памяти. Для замены блоков используется алгоритм LRU.

5.11. Повторите упражнение 5.10 в предположении, что при копировании в заполненное множество новый блок заменяет блок, использовавшийся последним.

5.12. В разделе 5.5.3 на примере программы, приведенной на рис. 5.19, был показан результат использования различных технологий отображения. Предположим, программа изменилась и теперь второй цикл обрабатывает элементы

в том же порядке, что и первый, то есть управляющая команда второго цикла такова:

for $i := 0$ **to** 9 **do**

Приведите для этой программы таблицы результатов, эквивалентные представленным на рис. 5.20–5.22. Какие выводы можно сделать из этого упражнения?

5.13. В побайтово адресуемом компьютере имеется маленький кэш данных, в котором может храниться восемь 32-разрядных слов. Каждый блок этого кэша состоит из одного 32-разрадяного слова. При выполнении программы процессор считывает данные по следующим шестнадцатеричным адресам:

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4

Эта последовательность операций чтения повторяется четыре раза.

а) Каким будет содержимое кэша с прямым отображением в конце каждого прохода по этому циклу. Вычислите для этого примера частоту попаданий. Предполагается, что первоначально кэш пуст.

б) Повторите упражнение 5.13, *а* для ассоциативного кэша, в котором используется алгоритм замены LRU.

в) Повторите упражнение 5.13, *а* для четырехканального множественно-ассоциативного кэша.

5.14. Повторите упражнение 5.13 с условием, что каждый блок кэша состоит из двух 32-разрядных слов. Для задания 5.13, *в* используйте двухканальный множественно-ассоциативный кэш.

5.15. Как влияет значение k в системе памяти с чередованием адресов, показанной на рис. 5.25, *б*, на размер блока кэш-памяти?

5.16. Во многих компьютерах размер блока кэш-памяти может составлять от 32 до 218 байт. Какие преимущества даст увеличение (или уменьшение) размера блока? А что изменится к худшему?

5.17. Рассмотрите эффективность чередования адресов по отношению к размеру блока кэша. С помощью вычислений, подобных проведенным в разделе 5.6.2, оцените возможное повышение производительности для блоков размером 16, 8 и 4 слова. Предполагается, что к каждому загруженному в кэш слову процессор обращается хотя бы один раз.

5.18. Предположим, что в компьютере имеется кэш-память L1 и L2, описанная в разделе 5.6.3. Размер блоков обоих кэшей составляет 8 слов. Предположим также, что частота попаданий в случае обоих кэшей одинакова и равна 0,95 для команд и 0,9 для данных. Время доступа к блоку из 8 слов в кэш-памяти первого и второго уровней, C_1 и C_2, составляет соответственно 1 и 10 циклов.

а) Каково среднее время доступа с точки зрения процессора, если используется память с чередованием адресов? Параметры доступа к памяти такие, как в разделе 5.6.1.

б) Каково среднее время доступа, если используется основная память без чередования адресов?

в) В чем преимущество чередования адресов?

5.19. Повторите упражнение 5.18 при условии, что блок кэша вмещает 4 слова. Оцените соответствующее значение C_2, если кэш второго уровня реализован на основе микросхем SRAM.

5.20. Рассмотрим следующую аналогию концепции кэширования. Мастер приходит в дом, чтобы починить отопительную систему. Он приносит с собой ящик с инструментами, которыми недавно выполнял подобные работы. Мастер опять будет использовать эти инструменты, пока ему не потребуются другие. Скорее всего, нужные инструменты найдутся в его машине, которая стоит возле дома. Но если их не окажется и там, придется ехать за ними в мастерскую.

Ящик с инструментами, машина и мастерская соответствуют кэшу L1, кэшу L2 и основной памяти компьютера. Насколько удачна эта аналогия? В чем она правильна, а в чем нет?

5.21. Массив 32-разрядных чисел размером 1024×1024 должен быть «нормализован» следующим образом. Определяется наибольший элемент каждого столбца и все элементы столбца делятся на это значение. Допустим, что каждая страница виртуальной памяти имеет размер 4 Кбайт, а для хранения данных в процессе вычислений выделен 1 Мбайт основной памяти. Предположим, что на загрузку страницы с диска в основную память при возникновении ошибки страницы уходит 40 мс.

а) Сколько страниц будет ошибочно загружено, если элементы массива хранятся в виртуальной памяти в порядке следования столбцов.

б) Сколько ошибок страниц произойдет в том случае, если элементы хранятся в порядке следования строк?

в) Оцените общее время, необходимое на выполнение описанной процедуры при условиях, указанных в заданиях 5.21, *а* и 5.21, *б*.

5.22. Рассмотрим компьютерную систему, в которой доступные страницы физической памяти разделяются между несколькими прикладными программами. Когда все выделенные для программы страницы заполняются и требуется новая страница, она должна заменить одну из резидентных. Операционная система управляет пересылкой страниц и динамически выделяет их прикладным программам. Предложите стратегию, которая могла бы использоваться операционной системой для минимизации общего количества операций пересылки страниц.

5.23. В компьютере с системой виртуальной памяти выполнение команды может быть прервано из-за ошибки страницы. Какую информацию о состоянии следует сохранить для того, чтобы впоследствии можно было возобновить реализацию этой команды? Учтите, что для переноса новой страницы в основную память необходима операция ПДП, для которой требуется выполнение других команд. Не проще ли отменить прерванную команду, а затем задать ее сначала? Возможно ли это?

5.24. Когда программа генерирует ссылку на страницу, отсутствующую в основной памяти, выполнение этой программы приостанавливается до тех пор, пока запрошенная страница не будет загружена в основную память. Какие трудности могут возникнуть в том случае, если команда находится на одной странице, а ее операнд на другой? Какими функциями должен обладать процессор, чтобы справиться с такой ситуацией?

5.25. В дисковом устройстве имеется 24 поверхности, на которые может записываться информация. Всего в нем 14000 цилиндров, на дорожке в среднем 400 секторов, а каждый сектор содержит 512 байт данных.

а) Определите максимальное количество байтов, которое можно сохранить на таком устройстве.

б) Какова скорость пересылки данных в байтах в секунду при скорости вращения магнитного диска 7200 об/мин?

в) Предложите схему задания адреса данных на диске при условии, что используются 32 разрядные слова.

5.26. Время поиска в сумме со временем задержки позиционирования при обращении к конкретному блоку данных на диске обычно значительно превышает время пересылки готовых данных. Рассмотрим процесс доступа к 3,5-дюймовому диску, описанному в разделе 5.9.1, для чтения или записи при условии, что средняя длина считываемого или записываемого блока равна 8 Кбайт.

а) Вычислите в процентах среднее время, уходящее на поиск и позиционирование, по отношению к общему времени доступа, если блоки расположены на диске случайным образом.

б) Повторите упражнение 5.26, *а* для ситуации, когда блоки расположены на диске так, что в 90 % случаев следующее обращение производится к данным, находящимся на том же цилиндре.

5.27. Среднее время поиска и задержка позиционирования составляют соответственно 6 и 3 мс. Скорость пересылки данных с диска и на диск составляет 30 Мбайт/с, и при каждом обращении считывается или записывается блок данных размером 8 Кбайт. Дисковый контроллер, контроллер ПДП, процессор и основная память соединены с одной 32-разрядной шиной. Передача данных по шине в основную память и из основной памяти занимает 10 нс.

а) Каково максимальное количество дисковых устройств, которые могут одновременно пересылать данные в основную память и из нее?

б) Сколько циклов основной памяти в процентах занимает дисковое устройство, которое в течение длительного периода времени выполняет пересылку независимых 8-килобайтовых данных?

5.28. Если в качестве вторичного запоминающего устройства для хранения программ и данных в системе виртуальной памяти используется магнитный диск, какие параметры этого диска должны быть учтены при выборе размера страницы?

5.29. Накопитель на магнитной ленте имеет следующие параметры:

Плотностью битов	2000 бит/см
Скорость движения ленты	80 см/с
Время на изменение направления перемотки	225 мс
Минимальное время прохода через пробел между записями	3 мс
Средняя длина записи	4000 символов

Оцените выигрыш во времени (в процентах), получаемый благодаря возможности читать записи в обоих направлениях. Предполагается, что доступ к записям осуществляется в произвольном порядке и что в среднем между двумя последовательно считываемыми записями расположено четыре записи.

Глава 6

Арифметика

- ✦ Высокоскоростные сумматоры с параллельным переносом
- ✦ Алгоритм Бута для умножения чисел со знаком
- ✦ Высокоскоростные умножители, осуществляющие параллельное сложение на основе алгоритма сложения с сохранением переноса
- ✦ Аппаратная реализация операции деления
- ✦ Представление чисел с плавающей запятой в стандартном формате IEEE и выполнение основных арифметических операций

Работа всех компьютеров основана на одной элементарной основной операции — сложении (или вычитании) двух чисел. Арифметические операции выполняются на уровне машинных команд. Эти команды, как и основные логические функции И, ИЛИ, НЕ и Исключающее ИЛИ, встроены в *арифметико-логическое устройство (АЛУ)* процессора. В настоящей главе мы познакомимся с логическими схемами, реализующими встроенные арифметические операции. Производительность процессора зависит от скорости выполнения операции сложения, а также операций умножения и деления, схемы которых сложнее. Вы узнаете, как в современных компьютерах на аппаратном уровне решаются проблемы повышения скорости арифметических операций.

В отличие от логических, арифметические операции сложно реализовывать на основе комбинационных схем: в первом случае выполняются независимые булевы (логические) операции над отдельными битами операнда, а во втором используются дополнительные сигналы переноса или отрицательного переноса.

В разделе 2.1 описывалось, как представлять двоичные числа со знаком, и было доказано, что при сложении и вычитании удобнее использовать систему дополнения до двух. На рис. 2.4 показан пример вычисления суммы двух n-разрядных чисел со знаком с помощью операции двоичного n-разрядного сложения. При этом знаковый бит обрабатывается так же, как и остальные. Другими словами, логические схемы для сложения двух двоичных чисел без знака могут быть использованы и для вычисления суммы чисел со знаком. В следующих двух разделах вы сможете познакомиться с логическими схемами для выполнения сложения и вычитания.

6.1. Сложение и вычитание чисел со знаком

На рис. 6.1 показана таблица истинности функций суммирования и переноса для поразрядного сложения чисел X и Y. На этом же рисунке приведены логические выражения для указанных функций и примеры сложения 4-разрядных беззнаковых чисел 7 и 6. Обратите внимание на то, что на каждом шаге процесса сложения требуется разряд переноса. Входной перенос в позицию i (он же выходной перенос из позиции $i - 1$) мы обозначаем как c_i.

Логическое выражение для s_i на рис. 6.1 можно реализовать с помощью 3-входового вентиля Исключающее ИЛИ, который на рис. 6.2, *а* включен в состав схемы, реализующей один шаг двоичного сложения. Функция выходного переноса, c_{i+1}, представлена в виде двухуровневой логической схемы И-ИЛИ. Для полной схемы одного шага сложения, называемой *полным сумматором (ПС)*, мы использовали отдельное обозначение.

При сложении двух n-разрядных двоичных чисел может использоваться каскадное соединение n блоков полных сумматоров (рис. 6.2, *б*). Поскольку переносы передаются от сумматора к сумматору, такая конфигурация называется *n-разрядным сумматором с последовательным переносом* (n-bit ripple-carry adder).

Входной перенос в позицию *самого младшего разряда* (Less Significant Bit, LSB) является удобным способом прибавления к числу единицы. Эта операция используется, например, при формировании дополнения числа до двух путем прибавления 1 к дополнению этого числа до единицы. Сигналы переноса применяются также при соединении k сумматоров для образования более сложного сумматора, способного складывать числа длиной kn бит (рис. 6.2, *в*).

x_i	y_i	Входной перенос c_i	Сумма s_i	Выходной перенос c_{i+1}
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

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i\overline{y}_i\overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Пример:

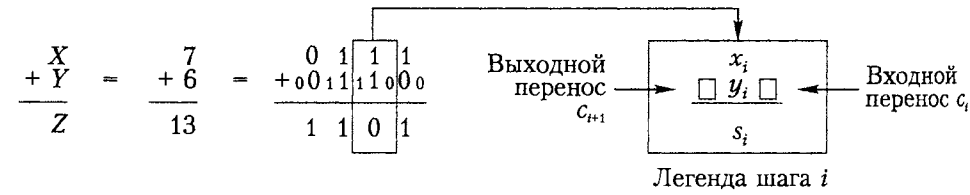
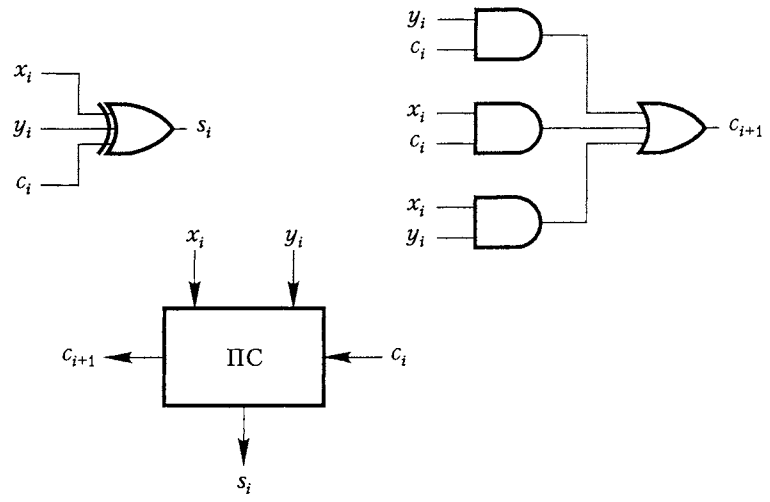
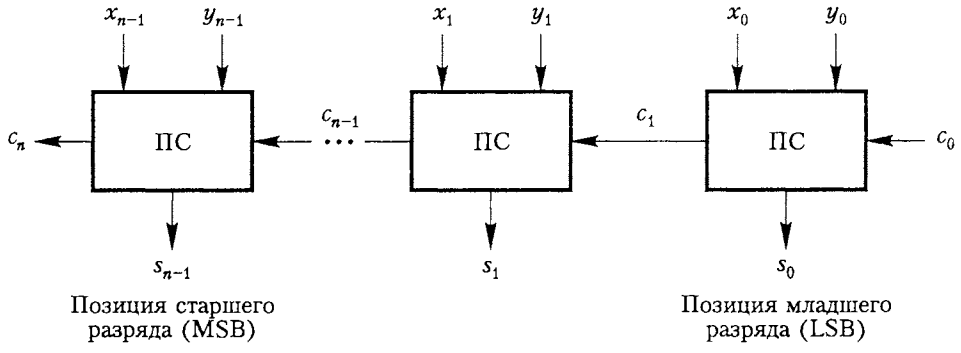


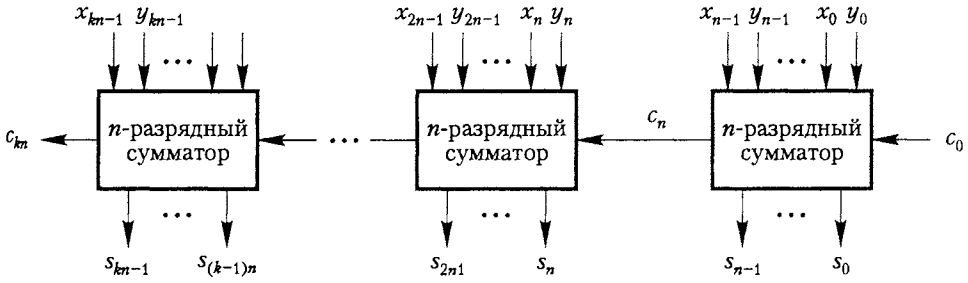
Рис. 6.1. Логическая спецификация одного шага двоичного сложения



а



б



в

Рис. 6.2. Логика сложения двоичных векторов: логика одного шага (*а*); n-разрядный сумматор с последовательным переносом (*б*); каскад n-разрядных сумматоров (*в*)

6.1.1. Логический блок сложения/вычитания

Для сложения дополнений чисел X и Y до двух, в которых разряды x_{n-1} и y_{n-1} содержат знак числа, может использоваться n-разрядный сумматор (рис. 6.2, *б*). В этом случае бит выходного переноса c_n не является частью ответа. Об арифметическом переполнении рассказывалось в разделе 2.1.4. Оно происходит только тогда, когда оба операнда имеют одинаковые знаки, отличные от знака результата. Таким образом, для обнаружения переполнения в n-разрядный сумматор нужно добавить специальную схему, реализующую такое логическое выражение:

$$\text{Переполнение} = x_{n-1}\bar{y}_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1}$$

Доказуемо, что переполнение возникает тогда, когда биты c_n и c_{n-1} имеют разные значения (см. упражнение 6.9). Поэтому для обнаружения переполнения в качестве альтернативы можно использовать выражение $c_n \oplus c_{n-1}$ с вентилем Исключающее ИЛИ.

Разность $X - Y$ вычисляется путем прибавления к числу X дополнения до двух числа Y. Логическая схема на рис. 6.3 может применяться как для сложения, так и для вычитания, что зависит от того, какое значение подано на ее входную управляющую линию сложения/вычитания. Если необходимо произвести сложение, на эту линию подается 0, на соответствующую группу входов сумматора — вектор Y, а на вход c_0 — значение 0. Когда управляющая линия сложения/вычитания устанавливается в 1, с помощью вентилей Исключающее ИЛИ формируется дополнение вектора Y до единицы (его побитовое дополнение), а на вход c_0 подается значение 1, завершающее формирование дополнения числа Y до двух. Напомним, что дополнение отрицательного числа до двух формируется так же, как и положительного числа. Схему на рис. 6.3 можно расширить за счет еще одного вентиля Исключающее ИЛИ, который выявляет условие переполнения $c_n \oplus c_{n-1}$.

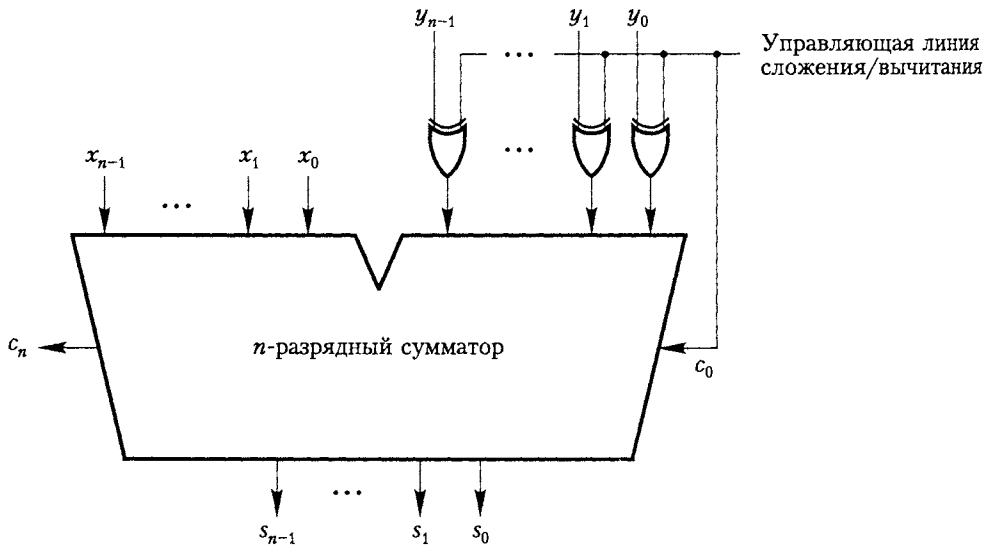


Рис. 6.3. Логическая схема двоичного сложения/вычитания

6.2. Архитектура быстродействующих сумматоров

Если в блоке сложения/вычитания, показанном на рис. 6.3, использовать n-разрядный сумматор с последовательным переносом, выходные сигналы на линиях s_0–s_{n-1} и c_n будут формироваться с очень большой задержкой. Чтобы решить, допустима ли такая задержка, следует учесть быстродействие других компонентов процессора, а также время пересылки данных регистров и кэш-памяти. Факторами, влияющими на задержку в схеме, состоящей из логических вентилей, являются технология ее создания и количество вентилей, через которые проходят сигналы на пути от входов к выходам. Задержка в любой комбинационной логической схеме на основе вентилей равна сумме задержек при прохождении сигналов через вентили по самому длинному пути в схеме. В случае n-разрядного сумматора с последовательным переносом этот путь пролегает от входов x_0, y_0 и c_0 в позиции LSB до выходов c_n и s_{n-1} в позиции *самого старшего разряда* (Most Significant Bit, MSB).

В логической схеме на рис. 6.2, *а* значение c_{n-1} формируется через $2(n-1)$ вентильные задержки, а на выход s_{n-1} правильное значение подается спустя одну задержку в вентиле Исключающее ИЛИ. Окончательный выходной перенос, c_n, формируется через $2n$ вентильных задержек. Таким образом, если блок сложения/вычитания, показанный на рис. 6.3, реализован в виде сумматора с последовательным переносом, все разряды суммы формируются через $2n$ вентильных задержек с учетом задержки в вентилях Исключающее ИЛИ на входе Y. Если контроль переполнения реализуется по формуле $c_n \oplus c_{n-1}$, индикатор переполнения формируется через $2n + 2$ вентильные задержки.

Существует два метода сокращения задержки в сумматорах. Первый заключается в подборе электронных элементов и электронных технологий, позволяющих предельно ускорить распространение переносов. Второй основан на оптимизации структуры, используемой для этого схемы. В следующем разделе применение второго метода рассматривается на примере схемы с параллельным переносом.

Построение такой схемы будет представлено в несколько упрощенном виде. На практике для реализации высокоскоростных сумматоров используется множество технологий. Это и электронные решения, направленные на ускорение распространения сигналов переноса, и различные варианты базовой схемы, описанной в следующем разделе.

6.2.1. Сложение с параллельным переносом

Для получения быстродействующей схемы сумматора нужно ускорить формирование сигналов переноса. Этого можно добиться за счет усовершенствования подсхемы распространения переносов.

Рассчитаем логические выражения для s_i (суммы) и c_{i+1} (выходного переноса) разряда i (рис. 6.1):

$$s_i = x_i \oplus y_i \oplus c_i$$

и

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Если во втором выражении вынести c_i за скобки:

$$c_{i+1} = x_i y_i + (x_i + y_i)c_i$$

то можно записать:

$$c_{i+1} = G_i + P_i c_i$$

где

$$G_i = x_i y_i,\ P_i = x_i + y_i$$

Выражения для G_i и P_i называют функциями *генерирования* и *распространения* разряда i. Если функция генерирования разряда i равна 1, то $c_i = 1$ независимо от входного переноса c_i. Так бывает, когда x_i и y_i равны 1. Функция распространения демонстрирует, что входной перенос будет преобразован в выходной, если $x_i = 1$ либо $y_i = 1$. Результаты функций G_i и P_i могут формироваться или независимо, или параллельно, спустя одну вентильную задержку после подачи на вход n-разрядного сумматора векторов X и Y. Схема каждой стадии сложения, соответствующей одному разряду слагаемых, содержит вентиль И для формирования функции G_i и вентиль ИЛИ для формирования функции P_i, а также 3-входовой вентиль Исключающее ИЛИ для формирования соответствующего разряда суммы s_i. Данную схему можно упростить, воспользовавшись тем, что эквивалентная функция распространения реализуется также на основе выражения $P_i = x_i \oplus y_i$, результат которого отличается от $P_i = x_i + y_i$ только когда $x_i = y_i = 1$. Однако в этом случае $G_i = 1$, так что совершенно не важно, равно P_i нулю или единице. Итак, при реализации 3-входной функции Исключающее ИЛИ в виде каскада из двух 2-входовых вентилей Исключающее ИЛИ для каждого шага сложения можно использовать базовую ячейку разряда, обозначенную на рис. 6.4, *а* как ЯР.

Если выразить c_i через переменные разряда $i - 1$ и подставить это выражение в предыдущую формулу, получится такое выражение:

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

Выполнив аналогичные подстановки до конца, получим следующий результат:

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-2} + \ldots + P_i P_{i-1} \ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 c_0 \qquad (6.1)$$

Все переносы могут быть готовы через три вентильные задержки после поступления входных сигналов X, Y и c_0, поскольку сигналы G_i и P_i формируются спустя всего одну вентильную задержку, а еще две вентильные задержки происходят в схемах И-ИЛИ для c_{i+1}. После очередной задержки в вентиле Исключающее ИЛИ будут готовы все разряды суммы. Таким образом, независимо от значения n длительность процесса сложения n-разрядных двоичных чисел составляет четыре вентильные задержки.

Теперь рассмотрим архитектуру 4-разрядного сумматора. Переносы в этом сумматоре реализуются так:

$$c_1 = G_0 + P_0 G_0$$
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

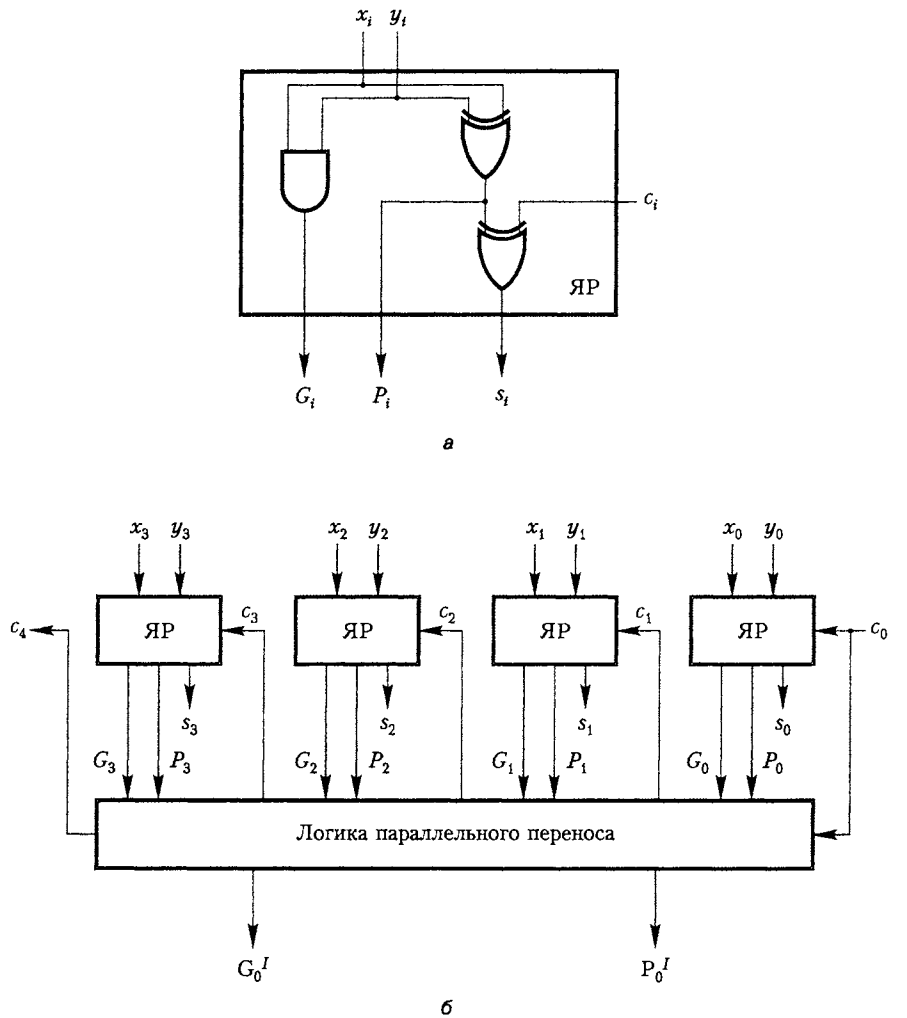


Рис. 6.4. Реализация 4-разрядного сумматора с параллельным переносом: ячейка для сложения одного разряда (*а*); схема 4-разрядного сумматора (*б*)

Полностью схема 4-разрядного сумматора приведена на рис. 6.4, *б*. Переносы реализованы в блоке, обозначенном как «Логика параллельного переноса». Этот сумматор называется *сумматором с параллельным переносом* (carry-lookahead adder). Формирование всех битов переноса выполняется в нем за три вентильные задержки, а вычисление полной суммы — за четыре. Для сравнения: 4-разрядный сумматор с последовательным переносом формирует значение s_3 за семь вентильных задержек, а значение c_4 — за восемь.

Предприняв попытку расширить сумматор с параллельным переносом, показанный на рис. 6.4, *б*, для более длинных операндов, мы столкнемся с проблемой ограниченной нагрузочной способности вентилей по входу. Из выражения 6.1 следует, что в последних вентилях И и ИЛИ, генерирующих значение c_{i+1}, коэффициент

объединения по входу должен быть равен $i + 2$. Для значения c_4 в 4-разрядном сумматоре коэффициент объединения по входу равен 5. Это почти предельное значение для реальных вентилей. Поэтому в рассматриваемом случае нельзя так просто расширить структуру сумматора в расчете на операнды большего размера. Сумматор большей разрядности можно сформировать путем каскадирования нескольких 4-разрядных сумматоров, что отражено на рис. 6.2, *в*.

Как же формируется 32-разрядный сумматор? Например, путем объединения восьми 4-разрядных сумматоров с параллельным переносом, как на рис. 6.2, *в*. Задержки при формировании разрядов суммы s_{31}, s_{30}, s_{29}, s_{28} и c_{32} в старшем 4-разрядном сумматоре этого каскада вычисляются следующим образом. Выходной перенос c_4 из младшего сумматора формируется через три вентильные задержки после того, как 32-разрядный сумматор получит входные операнды X, Y и c_0. Спустя две вентильные задержки появится перенос c_8 на выходе второго сумматора, затем — перенос c_{12} на выходе третьего сумматора и т. д. В конце концов на входе старшего 4-разрядного сумматора получаем c_{28}, на формирование которого с момента поступления входных операндов необходимо $(6 \times 2) + 3 = 15$ вентильных задержек. Еще через две задержки в старшем сумматоре будут готовы все переносы, включая c_{32}, а по прошествии еще одной задержки мы получим 4 разряда суммы. Итого — восемнадцать вентильных задержек. Сравните это значение с количеством задержек при формировании s_{31} и c_{32} в сумматоре с последовательным переносом — 63 и 64 задержки соответственно.

В следующем разделе рассказывается, как усовершенствовать описанную каскадную структуру, чтобы еще больше сократить общее количество задержек в сумматоре. Решение этой задачи состоит в параллельном формировании переносов c_4, c_8 и т. д. подобно тому, как формируются переносы c_1, c_2, c_3 и c_4 в 4-разрядном сумматоре с параллельным переносом.

Высокоуровневые функции генерирования и распространения

В 32-разрядном сумматоре из предыдущего раздела переносы c_4, c_8, c_{12} и т. д. последовательно распространяются от одного 4-разрядного суммирующего блока к другому с двумя задержками в каждом блоке (подобно тому, как отдельные переносы от разряда к разряду распространяются в сумматоре с последовательным переносом). Если использовать функции генерирования и распространения, действующие на уровне блоков, можно создать высокоуровневую схему параллельного переноса, обеспечивающую параллельное формирование переносов c_4, c_8, c_{12} и т. д.

На рис. 6.5 показан 16-разрядный сумматор, составленный из четырех 4-разрядных суммирующих блоков. Эти блоки реализуют новые выходные функции, обозначенные как G_k^I и P_k^I, где для первого 4-разрядного блока $k = 0$ (рис. 6.4, *б*), для второго 4-разрядного блока $k = 1$ и т. д. В первом блоке

$$P_0^I = P_3P_2P_1P_0$$

и

$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

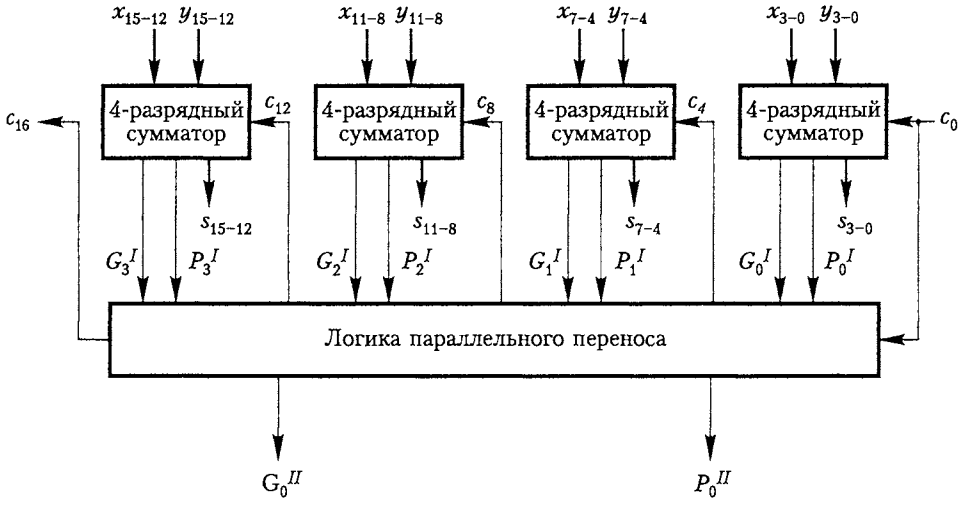


Рис. 6.5. 16-разрядный сумматор с параллельным переносом на основе 4-разрядных сумматоров (см. рис. 6.4, *б*)

Словесно это можно выразить так: функции G_i и P_i первого уровня определяют, имеется ли выходной перенос из разряда i (сгенерированный при сложении значений этого разряда или перешедший от предыдущего разряда), а функции второго уровня, G_k^I и P_k^I, устанавливают, есть ли выходной перенос из блока k. При наличии этих двух функций нет необходимости дожидаться последовательного распространения переноса через все 4-разрядные блоки. Перенос c_{16} формируется одной из схем параллельного переноса, представленной на рис. 6.5, с помощью такой функции:

$$c16 \quad G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I c0$$

Входные переносы 4-разрядных блоков организуются параллельно с помощью аналогичных кратких выражений. Эти выражения для переносов c_{16}, c_{12}, c_8 и c_4 имеют ту же форму, что и выражения для переносов c_4, c_3, c_2 и c_1, реализованные в схемах с параллельным переносом на рис. 6.4, *б*. Однако в схеме на рис. 6.5 переносы c_4, c_8, c_{12} и c_{16} не должны генерироваться внутри 4-разрядных суммирующих блоков, поскольку они генерируются уровнем выше, в блоке логики параллельного переноса. А теперь давайте посмотрим, какие задержки возникают при формировании выходных значений 16-разрядного сумматора с параллельным переносом.

Время готовности переносов, организуемых схемами параллельного переноса, на две вентильные задержки превышает время, необходимое для формирования результатов функций G_k^I и P_k^I. Для вычисления последних при наличии значений G_i и P_i требуются две и одна вентильные задержки. Таким образом, все переносы, генерируемые схемами параллельного переноса, будут готовы через пять вентильных задержек после поступления входных значений X, Y и c_0. Перенос c_{15} формируется внутри старшего 4-разрядного блока (рис. 6.5) через две задержки

после c_{12}, а по прошествии еще одной вентильной задержки получаем значение s_{15}. Итого, для формирования значения s_{15} нужно восемь вентильных задержек. Обратите внимание на то, что, если 16-разрядный сумматор формируется на основе последовательного каскада 4-разрядных суммирующих блоков с параллельным переносом, время вычисления значений c_{16} и s_{15} равно девяти и десяти вентильным задержкам. Сравните эти значения с теми, которые мы обсуждали, рассматривая схему на рис. 6.5 (вспомните, что в этом случае было пять и восемь вентильных задержек).

Для реализации 32-разрядного сумматора можно каскадировать два 16-разрядных суммирующих блока. В данной конфигурации выходной перенос c_{16} младшего блока становится входным переносом старшего. При этом задержка получается намного меньшей, чем в описанном выше 32-разрядном сумматоре, сформированном путем каскадирования восьми 4-разрядных сумматоров. Напомним, что тогда значение s_{31} было готово спустя восемнадцать вентильных задержек, а значение c_{32} — через семнадцать вентильных задержек. Давайте сравним эти значения с задержками в сумматоре, образованном на основе двух 16-разрядных суммирующих блоков. Как указывалось выше, перенос c_{16} из младшего блока формируется за пять вентильных задержек. Спустя две вентильные задержки готовы переносы c_{28} и c_{32} в старшем блоке, а через две задержки после c_{28} вычисляется значение c_{31}. Таким образом, для определения c_{31} требуется время, равное девяти вентильным задержкам, а для вычисления s_{31} — время, равное десяти задержкам. Итак, s_{31} и c_{32} формируются за десять и семь вентильных задержек, тогда как в сумматоре на основе каскада из восьми 4-разрядных сумматоров их формирование занимает восемнадцать и семнадцать вентильных задержек.

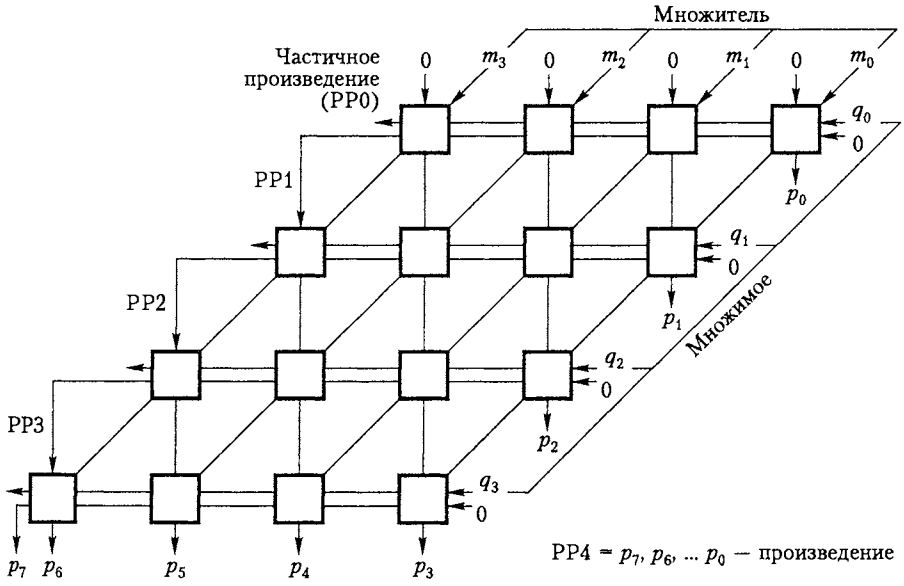
Принципы, которые использовались при формировании функций G_k^I и P_k^I на основе функций первого уровня G_i и P_i, можно применять для формирования функций третьего уровня G_k^{II} и P_k^{II} с использованием функций G_k^I и P_k^I. Две такие функции третьего уровня показаны на рис. 6.5 как выходы блока логики параллельного переноса. Из четырех 16-разрядных сумматоров, представленных на рис. 6.5, можно составить 64-разрядный сумматор, добавив еще один блок параллельного переноса, генерирующий переносы c_{16}, c_{32}, c_{48} и c_{64}. В таком сумматоре в общей сложности потребуется двенадцать вентильных задержек для формирования s_{63} и семь вентильных задержек для формирования c_{64}. Вычисляются они так же, как для 16-разрядного сумматора (см. упражнение 6.10).

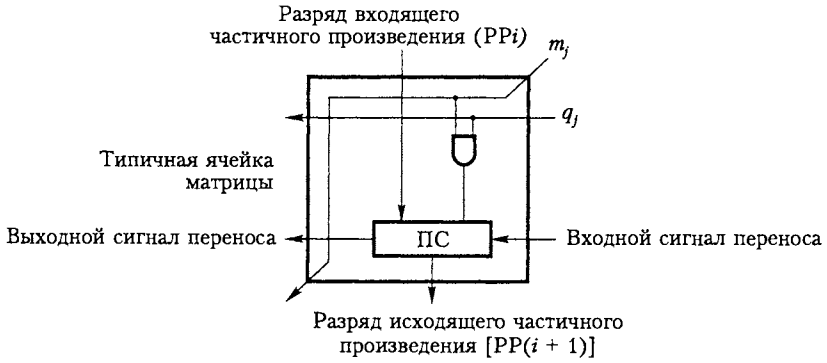
6.3. Умножение положительных чисел

На рис. 6.6, *а* вы видите классический алгоритм умножения целых чисел в двоичной системе счисления. Этот алгоритм применим к беззнаковым числам и положительным числам со знаком. Произведение двух n-разрядных чисел имеет длину не более $2n$, так что произведение двух 4-разрядных чисел в нашем примере помещается в 8 бит. В двоичной системе умножение первого числа на один разряд второго числа является простой операцией. Если этот разряд равен 1, первое число записывается в соответствующую позицию как есть, а если он равен 0, записываются нули, как в третьей строке нашего примера.

				1	1	0	1	(13) Множимое M
			×	1	0	1	1	(11) Множитель Q
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
	1	1	0	1				
1	0	0	0	1	1	1	1	(143) Произведение P

а





б

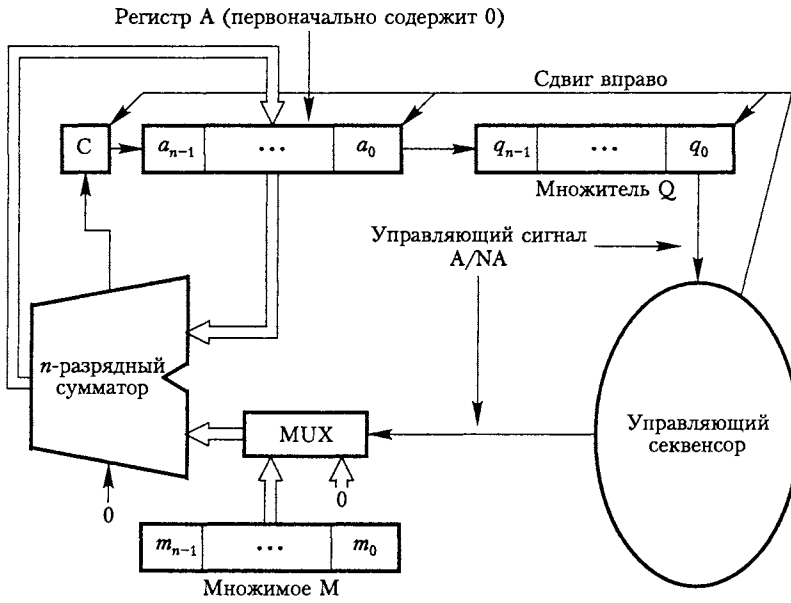
Рис. 6.6. Параллельный матричный умножитель положительных двоичных операндов: алгоритм умножения (*а*); матрица сумматоров (*б*)

Двоичное произведение положительных операндов можно реализовать в виде комбинационной двухмерной логической матрицы, показанной на рис. 6.6, *б*. Главным компонентом любой ячейки является полный сумматор ПС. Присутствующий в каждой ячейке вентиль И на основе значения бита множителя q_i определяет, добавляется ли ко входящему биту частичного произведения бит множимого m_j. Каждая строка i, где $0 \le i \le 3$, добавляет сдвинутое множимое ко входящему частичному произведению PPi для формирования выходного частичного произведения PP$(i + 1)$ при условии, что $q_i = 1$. Если же $q_i = 0$, PPi передается вниз без изменения. PP0 во всех позициях содержит нули, а PP4 представляет собой результирующее произведение. В каждой строке матрицы множимое сдвигается на одну позицию влево с помощью диагональных сигнальных линий.

В худшем случае задержка в этой схеме измеряется длиной пути от правого верхнего угла к левому нижнему углу матрицы, где формируется старший разряд произведения. Это ступенчатый путь, включающий по две ячейки с правого края каждой строки, а в конце — все ячейки нижней строки. Если предположить, что при прохождении сигналов от входов к выходам блока полного сумматора возникают две задержки в вентилях, весь этот путь для массива $n \times n$ сопровождается возникновением $6(n - 1) - 1$ вентильных задержек, с учетом задержки в первом вентиле И, который имеется в каждой ячейке (см. упражнение 6.12). В первой строке массива нужны только вентили И, поскольку входящее частичное произведение PP0 равно нулю. Мы учли это, рассчитывая задержку.

Операция умножения обычно входит в набор машинных команд процессора. В высокопроизводительных процессорах значительная часть микросхемы отведена под выполнение арифметических операций над целочисленными операндами и операндами с плавающей запятой. (Об операциях с плавающей запятой рассказывается далее в этой главе.) Хотя структура описанного комбинационного умножителя проста и понятна, для умножения чисел реального размера (32 или 64 разряда) он должен содержать очень много вентилей. Можно выполнять умножение иначе, сочетая технологию комбинационной матрицы (рис. 6.6) с последовательной технологией, в которой меньше комбинационной логики.

Простейший способ умножения основан на использовании имеющейся в АЛУ схемы сумматора, предназначенной для выполнения ряда последовательных шагов. На рис. 6.7, *а* демонстрируется, как взаимодействуют аппаратные элементы в процессе последовательного умножения. Для умножения используется один n-разрядный сумматор. Он n раз производит ту операцию, которую выполняет каждая строка сумматоров с последовательным переносом в схеме, предложенной вашему вниманию на рис. 6.6, б. В регистрах A и Q формируется частичное произведение PPi, а каждый бит q_i множителя, изначально хранящегося в регистре Q, генерирует сигнал A/NA (Add/Noadd). Этот сигнал управляет прибавлением множимого M к произведению PPi для получения значения PP$(i + 1)$. Все произведение вычисляется за n циклов. Длина частичного произведения на каждом шаге увеличивается на один разряд. С самого начала в регистре A содержится вектор PP0, состоящий из n нулей. Перенос из сумматора сохраняется в триггере C, показанном слева от регистра A.



а

	M		Q		
	1 1 0 1				Исходная конфигурация
0	0 0 0 0		1 0 1 1		
C	A		Q		
0	1 1 0 1		1 0 1 1	Сложение	Первый цикл
0	0 1 1 0		1 1 0 1	Сдвиг	
1	0 0 1 1		1 1 0 1	Сложение	Второй цикл
0	1 0 0 1		1 1 1 0	Сдвиг	
0	1 0 0 1		1 1 1 0	Нет сложения	Третий цикл
0	0 1 0 0		1 1 1 1	Сдвиг	
1	0 0 0 1		1 1 1 1	Сложение	Четвертый цикл
0	1 0 0 0		1 1 1 1	Сдвиг	
		Произведение			

б

Рис. 6.7. Последовательный двоичный умножитель: конфигурация (*а*); пример умножения (*б*)

Процесс умножения начинается с того, что множимое загружается в регистр M, множитель — в регистр Q, а C и A заполняются нулями. В конце каждого цикла C, A и Q сдвигаются на один разряд вправо, чтобы частичное произведение

увеличивалось по мере выдвижения множителя из регистра Q. В результате этого сдвига очередной разряд множителя q_i оказывается в позиции LSB регистра Q, которая отвечает за формирование сигнала A/NA, равного q_0 на первом цикле, q_1 на втором и т. д. При этом уже использованный бит регистра Q уничтожается сдвигом. Перенос из сумматора равен крайнему слева разряду PP(i + 1), должен быть сохранен в триггере C и сдвинут вправо вместе с содержимым регистров A и Q. После выполнения n циклов старшие разряды произведения оказываются в регистре A, а младшие — в регистре Q. На рис. 6.7, *б* показан процесс выполнения таким умножителем примера, приведенного на рис. 6.7, *а*.

Из приведенных схем видно, что для реализации команды умножения требуется гораздо больше времени, чем для выполнения команды сложения. Существует несколько технологий ускорения операции умножения; некоторые из них обсуждаются в следующих разделах.

6.4. Умножение чисел со знаком

Настало время поговорить об умножении операндов со знаком, которые представлены в формате дополнения до двух. Общая схема их умножения та же, что и в случае чисел без знака: путем многократного прибавления множимого либо набора нулей, что зависит от значения очередного разряда множителя, формируются частичные произведения.

Для начала предположим, что мы имеем положительный множитель и отрицательное множимое. Когда отрицательное множимое прибавляется к частичному произведению, по мере расширения частичного произведения значение знакового разряда должно расширяться влево. На рис. 6.8, например, 5-разрядное множимое со знаком, −13, умножается на +11, вследствие чего получаем 10-разрядное произведение — −143. Расширение знака множимого выделено полужирным шрифтом. Описанная схема может использоваться и для отрицательного множимого, если в ней обеспечено расширение знака частичных произведений.

Если множитель отрицателен, простейшее решение состоит в формировании дополнений до двух множимого и множителя и в последующем умножении их как положительных чисел. Это возможно благодаря тому, что при дополнении обоих операндов не меняется ни значение, ни знак произведения. В следующем разделе вы познакомитесь с еще одной технологией, *алгоритмом Бута*, которая позволяет одинаково обрабатывать положительные и отрицательные множители.

						1	0	0	1	1	(−13)
					×	0	1	0	1	1	(+11)
Полужирным шрифтом выделено расширение знака множимого	**1**	**1**	**1**	**1**	**1**	1	0	0	1	1	
	1	**1**	**1**	**1**	1	0	0	1	1		
	0	**0**	**0**	0	0	0	0	0			
	1	**1**	1	0	0	1	1				
	0	0	0	0	0	0					
	1	1	0	1	1	1	0	0	0	1	(−143)

Рис. 6.8. Расширение знака отрицательного множимого

6.4.1. Алгоритм Бута

В результате работы алгоритма Бута генерируется $2n$-разрядное произведение. При выполнении этой операции дополнения до двух как положительных, так и отрицательных n-разрядных операндов обрабатываются одинаковым образом. Чтобы лучше понять суть алгоритма Бута, рассмотрим операцию умножения с положительным множителем, содержащим один блок единиц: 0011110. Для получения произведения можно, как и в стандартной процедуре, сложить четыре значения множимого, сдвинутых влево на соответствующее количество разрядов. Однако число необходимых операций сократится, если множитель представлен в виде разности двух чисел:

$$\begin{array}{r} 0100000\ (32) \\ -\ 0000010\ (2) \\ \hline 0011110\ (30) \end{array}$$

Таким образом, для получения произведения можно сложить произведение множимого на 2^5 и дополнение до двух произведения множимого на 2^1. Описывать алгоритм удобнее, если представить множитель в виде 0 +1000 −10.

```
                                0  1  0  1  1  0  1
                                0  0 +1 +1 +1 +1  0
                                ------------------
                                0  0  0  0  0  0  0
                             0  1  0  1  1  0  1
                          0  1  0  1  1  0  1
                       0  1  0  1  1  0  1
                    0  1  0  1  1  0  1
                 0  0  0  0  0  0  0
              0  0  0  0  0  0  0
-------------------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0

                                0  1  0  1  1  0  1
                                0 +1  0  0  0 -1  0
                                ------------------
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  1  1  0  1  0  0  1  1  <-- Дополнение до двух множимого
0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  1  0  1  1  0  1
0  0  0  0  0  0  0  0
-------------------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

Рис. 6.9. Обычная схема умножения и алгоритм Бута

Если рассматривать общий случай, в алгоритме Бута множитель сканируется справа налево. При переходе от 0 до 1 к частичному произведению добавляется множимое, сдвинутое влево на соответствующее количество разрядов и умноженное на −1, а при переходе от 1 до 0 — произведение сдвинутого множимого и +1. На рис. 6.9 приведены примеры использования обычного алгоритма умножения и алгоритма Бута. Действие алгоритма Бута распространяется на любое количество блоков единиц в множителе, в том числе его можно применить в ситуации, когда блоком считается лишь одна единица. На рис. 6.10 предлагается еще один пример преобразования множителя. Если младший бит множителя равен 1, можно считать, что справа от него находится подразумеваемый 0. Алгоритм Бута подходит и для отрицательных множителей, о чем свидетельствует рис. 6.11.

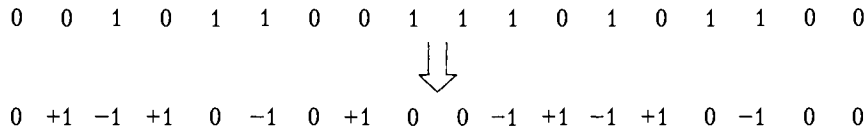


Рис. 6.10. Преобразование множителя в алгоритме Бута

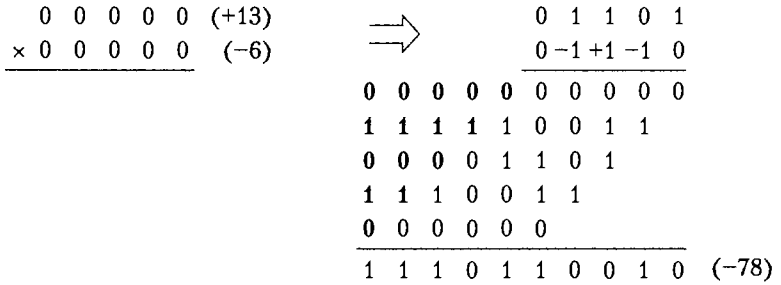


Рис. 6.11. Алгоритм Бута для отрицательного множителя

Давайте докажем, что алгоритм Бута корректно работает и при умножении отрицательных чисел. Для этого мы можем воспользоваться следующим свойством представления отрицательных чисел в системе дополнения до двух. Если крайний слева нуль отрицательного числа X находится в разряде k:

$$X = 11 \ldots 10x_{k-1} \ldots x_0$$

то значение X можно представить так:

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \ldots + x_0 \times 2^0$$

Правильность этого выражения $V(X)$ подтверждается следующим: если представить значение X как сумму двух чисел

$$\begin{array}{rl} & 11 \ldots 100000 \ldots 0 \\ + & 00 \ldots 00x_{k-1} \ldots x_0 \\ \hline X = & 11 \ldots 10x_{k-1} \ldots x_0 \end{array}$$

то верхним числом будет -2^{k+1} в формате дополнения до двух. Преобразованный множитель представляет собой второе число, к которому в позиции $k + 1$ добавляется -1. Например, множитель 110110 преобразуется в 0 −1 +1 0 −1 0.

Общая схема преобразования множителя в алгоритме Бута представлена на рис. 6.12. Преобразование 011 . . . 110 ⇒ + 100 . . . − 10 называется *переходом через единицы*. В этом термине отражен принцип действия алгоритма Бута, когда единицы множителя составляют несколько непрерывных блоков. Как видите, для формирования произведения нужно сложить всего несколько сдвинутых копий множимого, что значительно ускоряет процесс. В худшем случае, когда единицы и нули множителя чередуются, сложение приходится выполнять для каждого разряда множимого. В результате слагаемых получается даже больше, чем если бы не применялся алгоритм Бута. На рис. 6.13 показаны три варианта расположения нулей и единиц в множителе: худший, обычный и лучший.

Множитель		Версия множителя, заданная значением разряда i
Разряд i	Разряд $i - 1$	
0	0	0 × M
0	1	+1 × M
1	0	−1 × M
1	1	0 × M

Рис. 6.12. Таблица преобразования множителя в алгоритме Бута

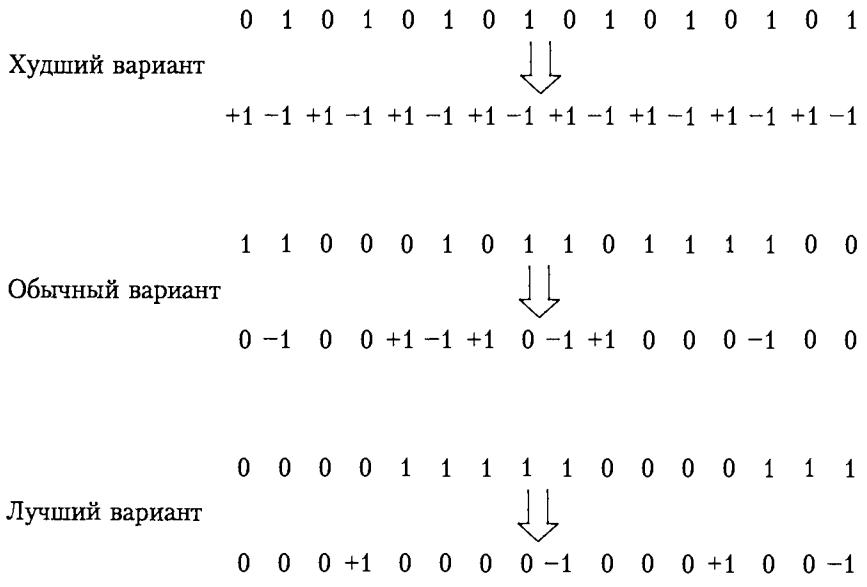


Рис. 6.13. Множители, преобразованные согласно алгоритму Бута

У алгоритма Бута есть две хорошие стороны. Он одинаково обрабатывает положительные и отрицательные числа, а если множитель содержит несколько больших блоков единиц, значительно ускоряется операция умножения. Ускорение за счет перехода через единицы зависит от значений данных. В среднем алгоритм Бута работает с той же скоростью, что и обычный алгоритм умножения.

6.5. Быстрое умножение

Настоящий раздел посвящен способам ускорения операции умножения. При использовании первого способа гарантируется, что максимальное количество слагаемых (версий множимого) для n-разрядных операндов не превысит $n/2$. Применение второго способа приводит к сокращению времени, затрачиваемого на операции сложения.

6.5.1. Перекодировка пар разрядов множителя

Благодаря технологии, называемой *перекодировкой пар разрядов*, вдвое уменьшается количество слагаемых, используемых для получения произведения. Ее можно назвать продолжением развития алгоритма Бута. Сначала множитель преобразовывается в соответствии с алгоритмом Бута, после чего его разряды разбиваются на пары. Каждая пара (+1 −1) эквивалентна паре (0 +1). Таким образом, вместо того чтобы прибавлять произведение (−1 × M), сдвинутое на i разрядов, к произведению (+1 × M), сдвинутому на (i + 1) разрядов, можно получить тот же результат, поместив в разряд i значение (+1 × M). Возможны и другие замены пар разрядов множителя. Например, пара (+1 0) эквивалентна паре (0 +2), пара (−1 +1) эквивалентна паре (0 −1) и т. д. Анализ разрядов множителя, преобразованного согласно алгоритму Бута, начиная справа, показывает, что их можно переписать таким образом, чтобы для каждой пары разрядов выполнялось не более одного сложения. На рис. 6.14, *а* приведен пример перекодировки пар разрядов множителя, который мы встречали на рис. 6.11, а на рис. 6.14, *б* воспроизведена таблица выбора версии множимого для каждого из возможных сочетаний разрядов множителя. На рис. 6.15 показано, как выполнить пример умножения, описанный на рис. 6.11, осуществляя перекодировку пар разрядов множителя.

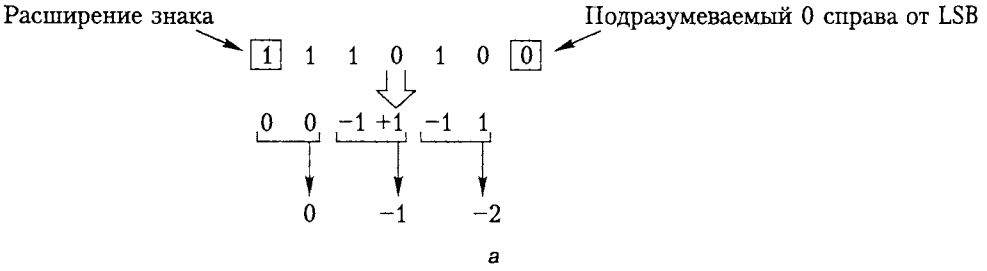


Рис. 6.14. Перекодировка пар разрядов множителя: после преобразования по алгоритму Бута (*а*); таблица выбора версий множимого (*б*)

Значения разрядов множителя		Значения разряда множителя справа	Версия множимого, выбираемая для разряда i
$i+1$	i	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

б

Рис. 6.14. (*продолжение*)

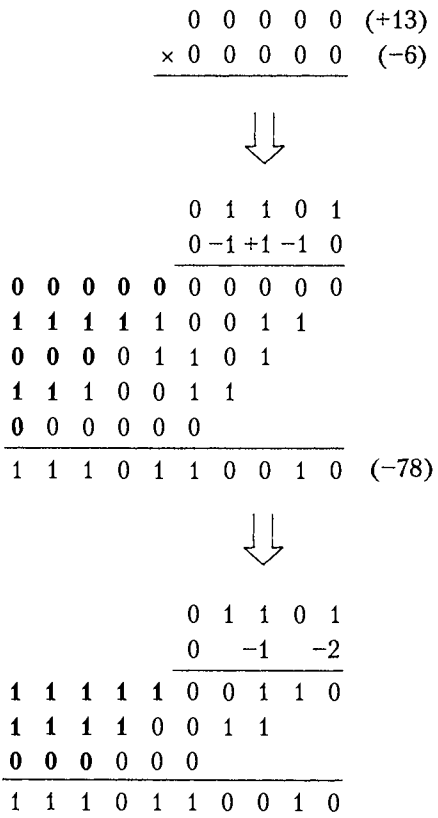


Рис. 6.15. Для умножения требуется только n/2 слагаемых

6.5.2. Сложение с сохранением переноса

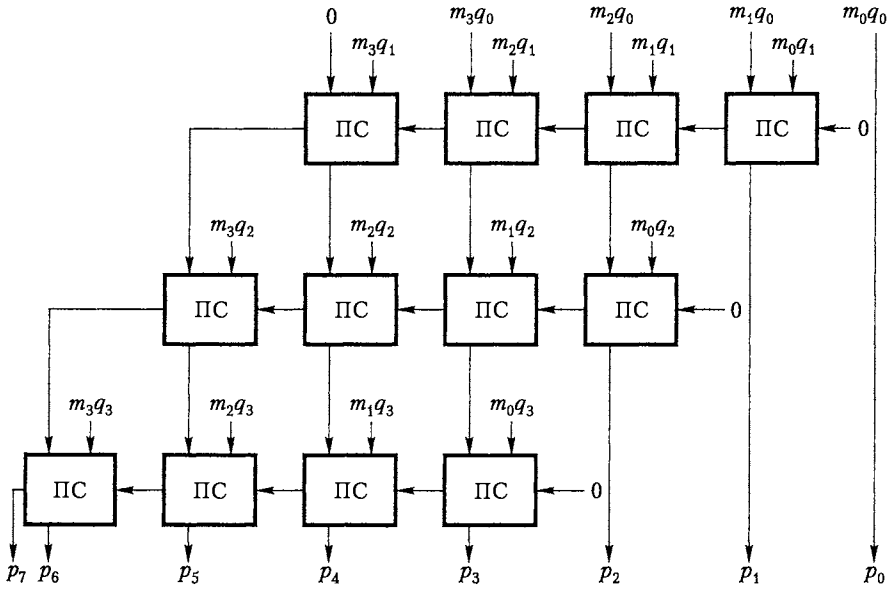
Умножение всегда выполняется путем сложения некоторого количества слагаемых. Этот процесс можно ускорить за счет *сложения с сохранением переноса* (Carry Save Addition, CSA). Рассмотрим матрицу умножения двух 4-разрядных чисел (рис. 6.16, *а*). Эта матрица знакома нам по рис. 6.6. В ее первой строке содержатся вентили И, с помощью которых генерируются битовые произведения m_3q_0, m_2q_0, m_1q_0 и m_0q_0.

Вместо того чтобы последовательно распространять переносы вдоль строк матрицы сумматоров, можно «сохранять» их и переносить в соответствующие позиции следующей строки, как показано на рис. 6.16, *б*. В результате у ряда полных сумматоров матрицы освобождается по одному входу. В частности, в первой строке освобождаются входы трех крайних слева сумматоров. Эти входы используются для битовых произведений третьего слагаемого: m_2q_2, m_1q_2 и m_0q_2. На два входа сумматоров во второй строке подаются выходные суммы и переносы из первой строки. Третий вход предназначается для битовых произведений m_2q_3, m_1q_3 и m_0q_3 четвертого слагаемого. Старшие битовые произведения m_3q_2 и m_3q_3 третьего и четвертого слагаемых подаются на оставшиеся свободные входы с левого края второй и третьей строк матрицы. Сохраненные биты переноса и разряды суммы из второй строки добавляются в третью строку, вследствие чего образуется полный набор разрядов произведения.

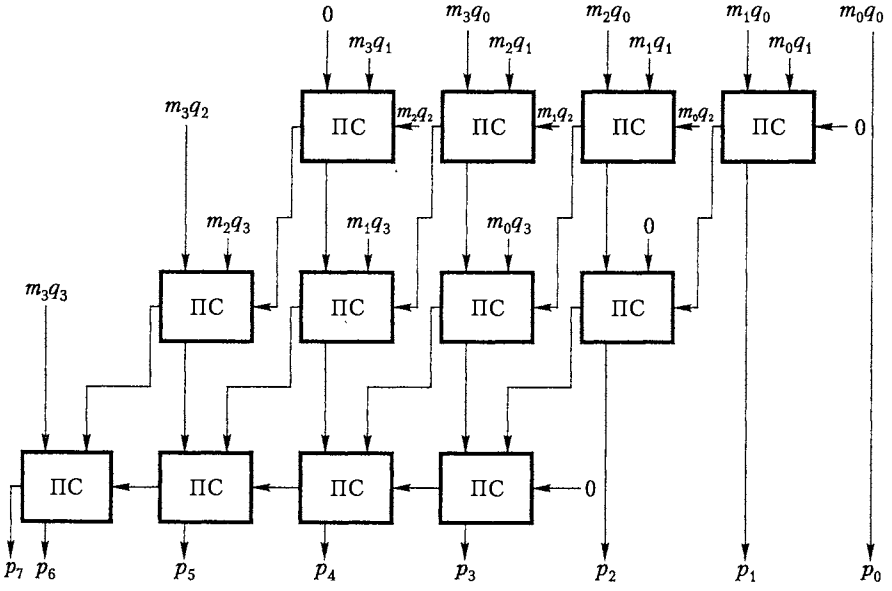
Полная задержка в матрице умножителя с сохранением переноса меньше, чем в обычной матрице с распространением переноса. Ускорение достигается за счет того, что выходные векторы каждой строки S и C формируются параллельно на протяжении одной задержки в полном сумматоре. В упражнении 6.22 вам предстоится случай самим оценить выигрыш во времени.

Описываемый ниже метод позволяет еще больше сократить задержку. Рассмотрим операцию суммирования большого количества слагаемых, выполняемую при умножении длинных операндов. Можно разбить слагаемые на тройки и для каждой из них произвести сложение с сохранением переноса, вследствие чего на протяжении одной задержки в полном сумматоре будет сформирован набор векторов S и C. Далее эти векторы снова объединятся в тройки, для которых тоже будет произведено сложение с сохранением переноса. Этот процесс продолжается до тех пор, пока не останется два вектора. Их сложение посредством сумматора с последовательным или параллельным переносом и дает результирующее произведение.

Обратимся к примеру. Предположим, требуется осуществить сложение шести сдвигаемых значений множимого при умножении двух 6-разрядных беззнаковых чисел. Все шесть разрядов множителя содержат единицы (рис. 6.17). Шесть слагаемых, A, B ... F, складываются множителем с сохранением переноса, как показано на рис. 6.18. Прямоугольники на этих двух рисунках выделяют соответствующие биты операндов, суммы и переноса. На рис. 6.19 схематически представлен трехуровневый процесс суммирования с сохранением переноса. Очевидно, что два последних вектора, S_4 и C_4, формируются за три задержки в полном сумматоре после подачи шести входных слагаемых на уровне 1. Последняя операция сложения векторов S_4 и C_4, в результате выполнения которой получаем конечное произведение, может быть произведена сумматором с последовательным или параллельным переносом.



а



б

Рис. 6.16. Матрицы для операции умножения 4-разрядных операндов M × Q = P: с последовательным распространением переноса (*а*); с сохранением переноса (*б*)

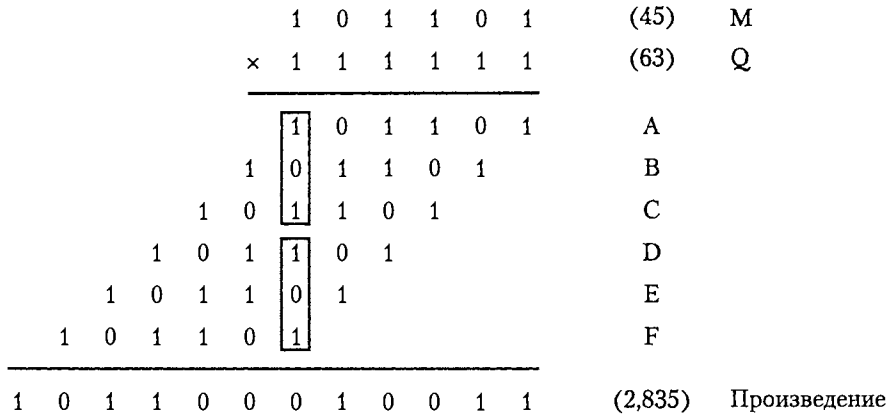


Рис. 6.17. Пример умножения с использованием операции сложения с сохранением переноса (для примера на рис. 6.18)

Давайте вычислим общее время, необходимое для умножения двух 6-разрядных чисел (рис. 6.18 и 6.19). После одной задержки в вентиле И, используемом для выбора слагаемых на основе значений разрядов множителя, все шесть слагаемых подаются на входы первого уровня CSA (сложения с сохранением переноса). Выходные векторы S_4 и C_4 третьего уровня CSA будут вычислены спустя шесть вентильных задержек (при двух вентильных задержках на каждом уровне CSA). Последние два вектора слагаются за восемь вентильных задержек, если применяется сумматор с параллельным переносом, как на рис. 6.5. Итого — пятнадцать вентильных задержек. Общее время, необходимое для умножения с использованием матрицы $n \times n$, подобной изображенной на рис. 6.6, равно $6\ (n - 1) - 1$. Таким образом, если применяется матрица 6×6, умножение выполняется за двадцать девять вентильных задержек. Как видите, в результате использования алгоритма сложения с сохранением переноса и сложения последних двух векторов с параллельным переносом продолжительность операции умножения сокращается вдвое.

Чем больше слагаемых, тем значительнее экономия времени. Например, для сложения 32-разрядных чисел по методу с сохранением переноса (рис. 6.19) требуется только восемь уровней CSA (до последней операции сложения). В общем случае, чтобы свести k слагаемых к двум векторам, при сложении которых получается конечная сумма, необходимо около $1{,}7 \log_2 k - 1{,}7$ уровней CSA (см. упражнение 6.23). Для сложения двух конечных векторов можно использовать 64-разрядный сумматор с параллельным переносом. Общую задержку при умножении 32×32 составляют: одна вентильная задержка в начальной группе вентилей И, на выходах которых получается 32 слагаемых; шестнадцать вентильных задержек для восьми уровней CSA; двенадцать вентильных задержек для самого длинного пути (до s_{63}) через 64-разрядный сумматор. Итого — двадцать вентильных задержек. Сравните: обычный матричный умножитель 32×32 генерирует последний бит произведения за сто восемьдесят пять вентильных задержек.

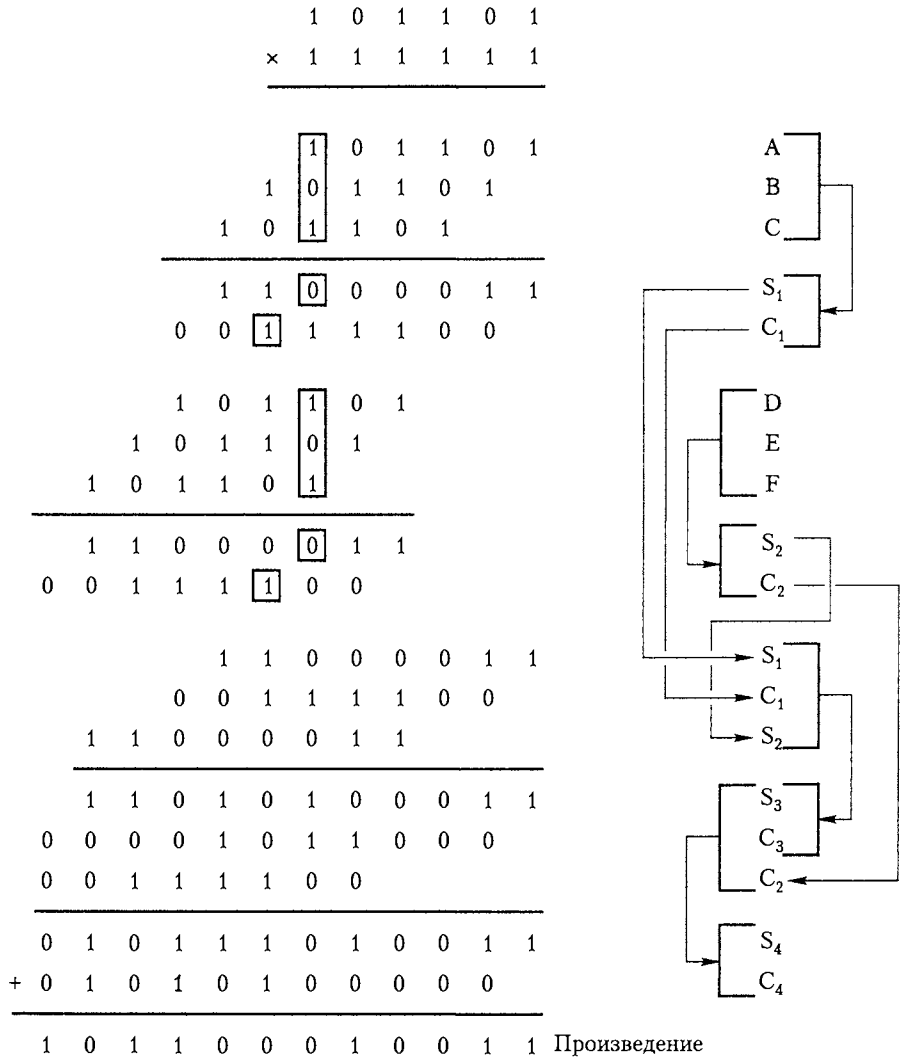


Рис. 6.18. Пример выполнения умножения с использованием операции сложения с сохранением переноса (для примера на рис. 6.17)

Обсуждая алгоритм сложения с сохранением переноса, мы опустили некоторые существенные моменты. Прежде всего отметим, что на случай отрицательных слагаемых логика CSA должна предусматривать расширение знака (как в алгоритме Бута). Расширение знака до полной длины произведения (удвоенной длины операнда) не обязательно. На каждом уровне CSA достаточно расширить знак на несколько разрядов. Еще один момент. Мы предполагаем, что для сложения двух последних векторов S и C в выражении $n \times n$ нужен $2n$-разрядный сумматор с параллельным переносом. В процессе выполнения этой операции на самом деле складывается несколько меньшее количество разрядов, поскольку часть

младших разрядов произведения к этому времени уже определена. Но это не играет решающей роли, и в целом произведенный анализ времени умножения остается правильным. Последнее: для выполнения операции умножения с использованием матрицы $n \times n$ необходимо n слагаемых. Однако после перекодировки пар разрядов количество слагаемых уменьшается до $n/2$. В результате и число уровней CSA сокращается с $1{,}7 \log_2 n - 1{,}7$ до $1{,}7 \log_2 n - 3{,}4$.

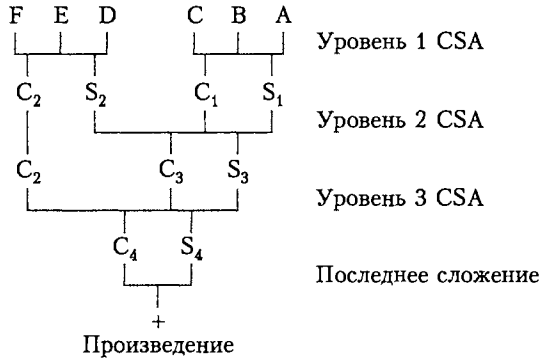


Рис. 6.19. Схематическое представление операций сложения (рис. 6.18) с сохранением переноса

Ускоренное умножение: итоги

Вы многое узнали о технологии высокоскоростного умножения. Давайте подведем итоги. Перекодировка пар разрядов (результат развития идеи, заложенной в алгоритм Бута) позволяет вдвое сократить количество слагаемых. Количество этих слагаемых вы можете уменьшить до двух путем выполнения относительно небольшого количества операций сложения с сохранением переноса. Результирующее произведение получается путем сложения этих двух слагаемых в сумматоре с параллельным переносом. Все три технологии — перекодировка пар разрядов множителя, сложение с сохранением переноса и сложение с параллельным переносом — повсеместно применяются конструкторами высокопроизводительных процессоров для сокращения времени выполнения умножения.

6.6. Целочисленное деление

В разделе 6.4, изучая процедуру умножения положительных чисел, мы сравнивали процесс умножения вручную с процессом умножения с помощью логических схем. Такой же подход будет избран и при описании операций целочисленного деления. Сначала мы обстоятельно рассмотрим деление положительных чисел, а затем поговорим об особенностях деления чисел со знаком.

На рис. 6.20 приведены примеры деления одних и тех же чисел в десятичной и двоичной системах счисления. Начнем с десятичных чисел. Цифру 2 в частном получаем следующим образом. Сначала предпринимается попытка разделить 2 на 13 — не делится. Тогда мы делим 27 на 13. Результат — 2 ($13 \times 2 = 26$) и 1 в остатке. К единице дописываем 4 из делимого и продолжаем процесс — делим 14 на

13, вследствие чего получаем 1 (переносим в частное) и 1 в остатке. Двоичное умножение выполняется аналогичным образом, но является более простым, поскольку в частном могут быть только единицы и нули.

```
   21                  10101
13 ) 274          1101 ) 100010010
  26                 1101
  14                 10000
  13                  1101
   1                  1110
                      1101
                         1
```

Рис. 6.20. Примеры операции деления

Алгоритм деления десятичных чисел реализуется следующим образом. Делитель располагается под старшими разрядами делимого, после чего выполняется вычитание. Если остаток положителен или равен нулю, в частное записывается 1, а остаток дополняется следующим разрядом делимого. Делитель перемещается в новую позицию, и вычитание повторяется. Если же остаток отрицательный, в частное записывается 0, делимое восстанавливается, для чего к нему прибавляется делитель, который смещается для проведения следующего вычитания.

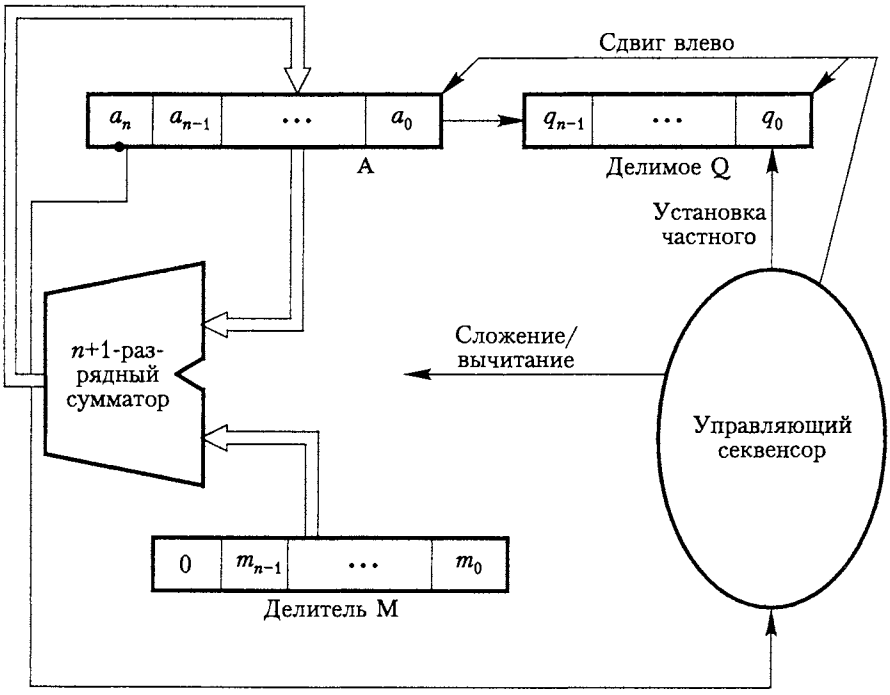


Рис. 6.21. Схема для двоичного деления

Деление с восстановлением

На рис. 6.21 представлена схема, по которой реализуется *деление с восстановлением* (restoring division). Обратите внимание на то, как она похожа на схему умножителя, приведенную на рис. 6.7. В начале операции n-разрядный положительный делитель загружается в регистр M, а n-разрядное положительное делимое — в регистр Q. Регистр A устанавливается в 0. По завершении операции деления n-разрядное частное находится в регистре Q, а остаток — в регистре A. Вычитание производится в соответствии с арифметикой дополнения до двух. В процессе вычитания лишние разряды с левого края регистров A и M заполняются значением знакового разряда. Алгоритм деления с восстановлением включает следующие шаги, которые повторяются n раз.

1. Сдвиг A и Q влево на одну двоичную позицию.
2. Вычитание M из A и помещение результата в A.
3. Если знак A равен 1, установка q_0 в 0 и добавление M к A (то есть восстановление A); иначе q_0 устанавливается в 1.

На рис. 6.22 приведен пример деления с 4-разрядным делимым, реализуемый схемой, которая представлена на рис. 6.21.

Деление без восстановления

Алгоритм деления с восстановлением можно усовершенствовать таким образом, чтобы в случае неудачного вычитания не приходилось восстанавливать значение A. Вычитание считается неудачным, если его результат отрицателен. Рассмотрим последовательность действий, совершаемых после операции вычитания в приведенном выше алгоритме. Если A положительно, мы сдвигаем его влево и вычитаем из него M, то есть выполняем операцию 2A − M. Если A отрицательно, мы восстанавливаем его с помощью операции A + M и сдвигаем влево, после чего вычитаем из него M, что равнозначно 2A + M. Впоследствии бит q_0 устанавливается в 0 или 1. На этой основе можно разработать следующий алгоритм *деления без восстановления*.

Шаг 1: Перечисленные ниже действия выполняются n раз.

1. Если знак A равен 0, сдвинуть A и Q на один разряд влево, а затем вычесть M из A; иначе сдвинуть A и Q влево и прибавить M к A.
2. Если знак A равен 0, установить q_0 в 1; иначе установить q_0 в 0.

Шаг 2: Если знак A равен 1, прибавить M к A.

Шаг 2 необходим для того, чтобы по завершении n циклов шага 1 в A остался положительный остаток. Логическая схема, приведенная на рис. 6.21, подходит и для описываемого алгоритма. Обратите внимание на то, что выполнять операции восстановления больше не нужно и что на каждом шаге цикла производится одна операция сложения или вычитания. На рис. 6.23 показано, как выполнить пример, приведенный на рис. 6.22, с помощью алгоритма деления без восстановления.

```
      10
11 ) 1000
     11
     10
```

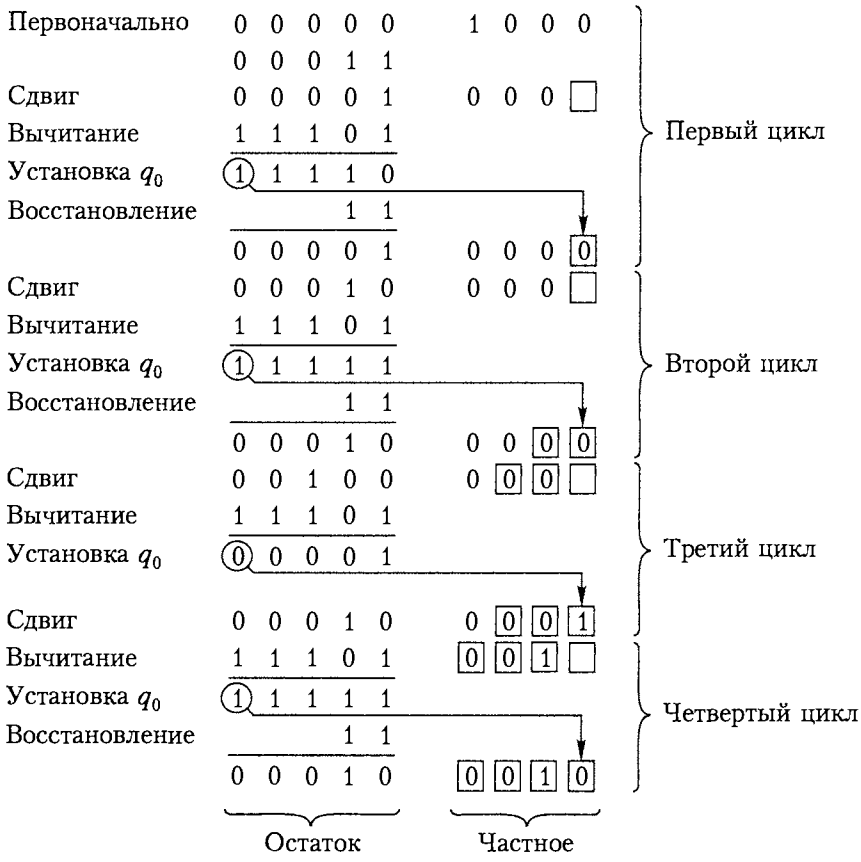


Рис. 6.22. Пример деления с восстановлением

К сожалению, простого алгоритма для непосредственного деления операндов со знаком, подобного алгоритмам умножения, не существует. Поэтому перед делением операнды со знаком должны быть преобразованы в положительные числа. После применения одного из описанных алгоритмов деления результат преобразуется в соответствующее значение со знаком. Разумеется, при этом учитываются знаки делимого и делителя.

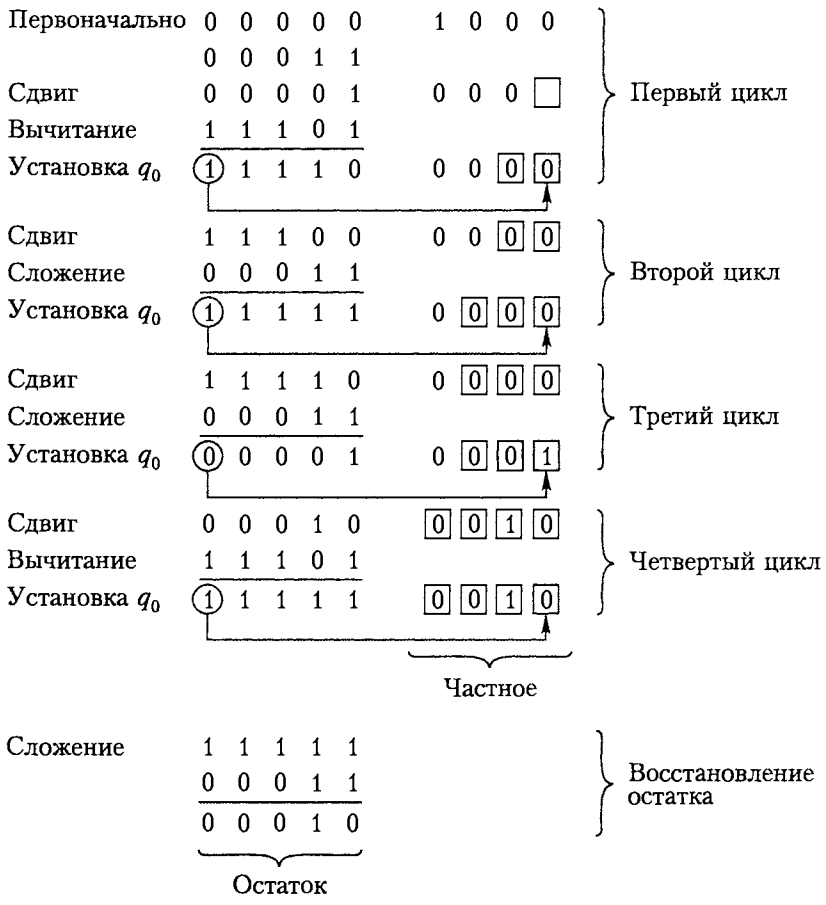


Рис. 6.23. Пример деления без восстановления

6.7. Обработка чисел с плавающей запятой

До сих пор речь шла только о числах с фиксированной запятой и целочисленных значениях, то есть таких, при обработке которых предполагается, что двоичная запятая находится справа от числа. Можно также считать, что двоичная запятая располагается справа от знакового разряда, определяя дробное значение. В системе дополнения до двух значение числа со знаком F, представленного n-разрядной двоичной дробью

$$B = b_0,b_{-1}b_{-2}\ldots b_{-(n-1)}$$

определяется функцией

$$F(B) = -b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \ldots + b_{-(n-1)} \times 2^{-(n-1)}$$

где значение F лежит в диапазоне

$$-1 \le F \le 1 - 2^{-(n-1)}$$

Рассмотрим диапазон значений, которые можно представить в 32-разрядном формате с фиксированной запятой. Если интерпретировать их как целые числа, получается диапазон от 0 до $\pm 2{,}15 \times 10^9$, а если как дроби — диапазон от $\pm 4.55 \times 10^{-10}$ до ± 1. Ни одного из этих диапазонов не достаточно для научных вычислений, в которых могут использоваться такие параметры, как число Авогадро ($6{,}0247 \times 10^{23}$ моль$^{-1}$) и константа Планка ($6{,}6254 \times 10^{-27}$ эрг·с). Необходим такой формат, который подходил бы и для очень больших целых, и для очень маленьких дробных чисел. Для этого компьютер должен уметь представлять числа и оперировать ими таким образом, чтобы позиция двоичной запятой была переменной и автоматически изменялась в процессе вычислений. Такие числа называют *числами с плавающей запятой*. Напомним, что в числах с фиксированной запятой двоичная запятая всегда располагается в одной и той же позиции.

Поскольку в числе с плавающей запятой позиция двоичной запятой переменная, она должна быть явно задана в представлении числа. Так, в хорошо знакомом вам научном десятичном формате числа могут записываться как $6{,}0247 \times 10^{23}$, $6{,}6254 \times 10^{-27}$, $-1{,}0341 \times 10^2$, $-7{,}3000 \times 10^{-14}$ и т. д. Говорят, что в этих числах по пять *значащих цифр*. *Масштабные множители* (10^{23}, 10^{-27} и т. д.) указывают позицию десятичной запятой по отношению к значащим цифрам. Числа, в которых десятичная запятая расположена справа от первой (ненулевой) значащей цифры, называют *нормализованными*. Основание масштабного множителя (10), которое является фиксированным, в машинном представлении чисел с плавающей запятой можно не задавать. Итак, число с плавающей запятой должно содержать знак, значащие цифры и показатель степени 10 в масштабном множителе. Теперь можно дать точное определение машинного представления числа с плавающей запятой: оно состоит из знака, строки значащих цифр, называемой *мантиссой*, и показателя степени (фиксированного основания), называемого *порядком*.

6.7.1. Стандарт IEEE для чисел с плавающей запятой

Сначала рассмотрим в общих чертах представление чисел с плавающей запятой в десятичной системе счисления, а затем соотнесем это представление с двоичным. Итак, числа с плавающей запятой удобно записывать следующим образом:

$$\pm X_1,X_2\ X_3\ X_4\ X_5\ X_6\ X_7 \times 10^{Y_1Y_2}$$

Здесь X_i и Y_i — это десятичные цифры. Такого количества значащих цифр (7) и диапазона значений порядка (± 99) достаточно для научных расчетов. Мантиссу и порядок из этих диапазонов можно перевести в двоичное представление, умещающееся в 32 разряда, то есть в стандартное компьютерное слово. Мантисса длиной 24 бита может представлять десятичное число из 7 цифр, а 8-битовый порядок подразумеваемого основания 2 позволяет представить достаточно широкий диапазон значений масштабных множителей. Для знака числа необходим

один бит. Поскольку ведущий бит нормализованной мантиссы обязательно должен быть равен 1, его можно не включать в представление, за счет чего и освобождается один бит для знака. Таким образом, 32 бита позволяют представить достаточно широкий диапазон чисел с плавающей запятой.

Описанный стандарт представления чисел с плавающей запятой в 32-разрядном формате разработан и детально специфицирован Институтом инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers, IEEE). В нем определены и представление чисел, и правила выполнения четырех базовых арифметических операций. На рис. 6.24, *а* продемонстрировано 32-разрядное представление чисел с плавающей запятой. Знак числа задается в первом разряде, за ним следует представление порядка (по основанию 2). Вместо числа со знаком в поле порядка *E* хранится целое число без знака E' = E + 127. Этот формат называется *форматом с избытком 127*. Таким образом, E' входит в диапазон $0 \le E' \le 255$. Граничные значения указанного диапазона, 0 и 255, употребляются для представления описанных ниже специальных значений. Для обычных (нормальных) значений E' лежит в пределах $1 \le E' \le 254$. Это означает, что реальный порядок, *E*, находится в диапазоне $-126 \le E \le 127$. Представление порядка в формате *с избытком x* упрощает сравнение относительного размера двух чисел с плавающей запятой (см. упражнение 6.27).

Последние 23 разряда числа представляют мантиссу. Поскольку числа задаются в нормализованном виде, старший бит мантиссы всегда равен 1. Этот бит не указывается явно; подразумевается, что он располагается слева от двоичной запятой. Таким образом, 23 бита в поле *M* соответствуют дробной части мантиссы, то есть разрядам справа от двоичной запятой. Пример числа с плавающей запятой одинарной точности приведен на рис. 6.24, *б*.

Стандартное 32-разрядное представление (рис. 6.24, *а*) называется представлением с *одинарной точностью*, поскольку занимает одно 32-разрядное слово. Оно позволяет представлять масштабные множители в диапазоне от 2^{-126} до 2^{+127}, что приблизительно равно $10^{\pm 38}$. Значение 24-разрядной мантиссы обеспечивает примерно ту же точность, что и семизначное десятичное значение. Для достижения большей точности и увеличения диапазона чисел с плавающей запятой стандарт IEEE определяет формат *двойной точности* (рис. 6.24, *в*). В нем расширены диапазоны значений мантиссы и порядка. Так, 11-разрядный порядок в формате с избытком 1023 E' лежит в диапазоне $1 \le E' \le 2046$ для обычных значений, а значения 0 и 2047 употребляются для представления специальных символов. Следовательно, действительный порядок *E* лежит в диапазоне $-1022 \le E \le 1023$, позволяющем представить масштабные множители от 2^{-1022} до 2^{1023} (то есть приблизительно $10^{\pm 308}$). Значения 53-разрядной мантиссы обеспечивают практически ту же точность, что и 16 десятичных цифр.

Чтобы компьютер соответствовал стандарту IEEE, он должен поддерживать как минимум представление чисел с плавающей запятой одинарной точности. Представление двойной точности необязательно. Этот стандарт определяет еще несколько необязательных расширенных версий обоих форматов. Они предназначены для повышения точности и порядка представления промежуточных результатов последовательных вычислений. Например, внутреннее произведение

двух векторов можно вычислить путем накопления суммы произведений с расширенной точностью. Входные значения имеют стандартную точность, одинарную или двойную; результат округляется до той же точности. Благодаря расширенным форматам можно сократить погрешность округления, которая накапливается при многократных однотипных вычислениях. Кроме того, расширенные форматы повышают точность вычисления таких элементарных функций, как синус, косинус и т. п. Наряду с четырьмя базовыми арифметическими операциями стандарт IEEE определяет операции вычисления остатка от деления, квадратного корня, преобразования из двоичного кода в десятичный и наоборот.

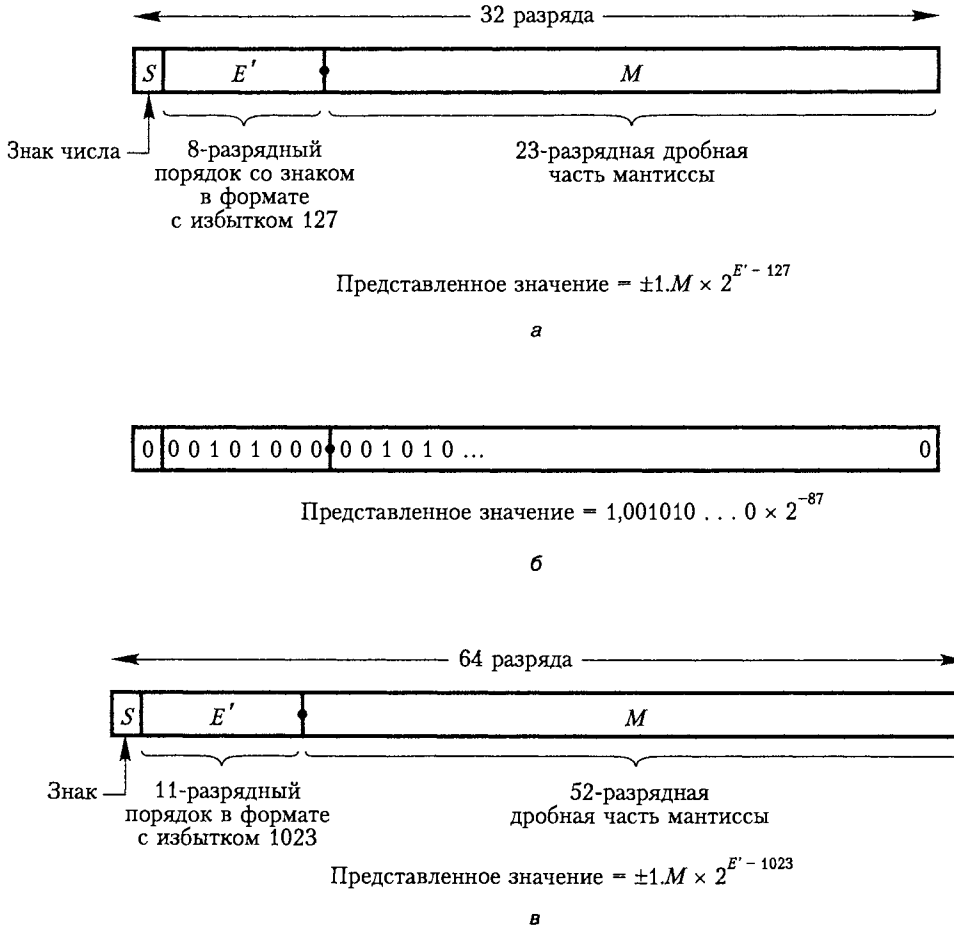


Рис. 6.24. Представления чисел с плавающей запятой, определенные в стандарте IEEE: формат числа с одинарной точностью (*а*); пример числа с одинарной точностью (*б*); формат числа с двойной точностью (*в*)

Существует два момента, касающихся чисел с плавающей запятой, которые заслуживают особого внимания. Во-первых, если число не нормализовано, его

всегда можно привести к нормальной форме, сдвинув дробную часть и соответствующим образом изменив порядок. На рис. 6.25 вы видите ненормализованное значение 0,0010110... × 2^9 и его нормализованное представление 1,0110... × 2^6. Поскольку масштабный множитель представлен в формате 2^i, сдвиг мантиссы на один разряд вправо или влево компенсируется увеличением или уменьшением порядка на единицу. Во-вторых, в ходе вычислений может быть сгенерировано число, выходящее за рамки диапазона нормальных чисел. Если точность одинарная, это означает, что для представления нормализованного числа требуется порядок менее −126 или более +127. В первом случае говорят о *потере значимости* или *отрицательном переполнении* (underflow), а во втором — о *переполнении* (overflow). И потеря значимости, и переполнение являются арифметическими исключениями, о которых мы поговорим чуть позже.

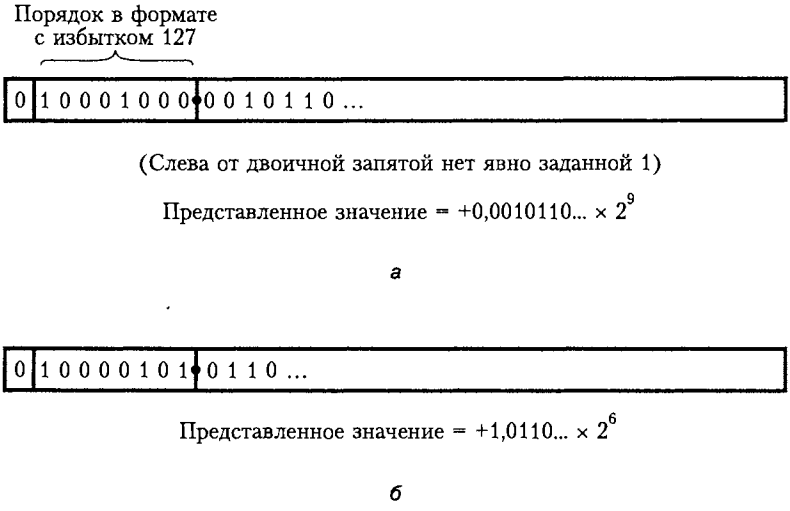


Рис. 6.25. Число с плавающей запятой в формате IEEE одинарной точности: ненормализованное (*а*); нормализованное (*б*)

Специальные значения

Граничные значения 0 и 255 порядка E' в формате с избытком 127 используются для представления специальных значений. Если $E' = 0$ и дробная часть мантиссы M равна нулю, значит, представлено точное значение 0. Порядок $E' = 255$ и мантисса $M = 0$ представляют значение ∞, где ∞ — результат деления нормального числа на нуль. Для представления этих значений обычно применяется и знаковый разряд, например: ±0 и ±∞.

Значения $E' = 0$ и $M \neq 0$ соответствуют представлению *анормальных* чисел. Это числа $\pm 0,M \times 2^{-126}$, которые меньше самого маленького числа. У них отсутствует подразумеваемая единица слева от двоичной запятой, а M представляет собой любую ненулевую 23-разрядную дробную часть числа. Анормальные числа предназначены для случаев, когда возможна *постепенная потеря значимости*; они расширяют диапазон представляемых чисел и могут быть полезны при работе

с очень маленькими числами. Когда $E' = 255$ и $M \neq 0$, представленное значение называется *Not a Number* (*NaN*). Значение *NaN* является результатом выполнения недопустимой операции, такой как 0/0 или $\sqrt{-1}$.

Исключения

Согласно стандарту IEEE, если в ходе работы произойдет потеря значимости, переполнение или деление на нуль, встретится условие inexact либо invalid, процессор должен установить флаг *исключения*. О первых трех условиях исключений мы уже упоминали. *Inexact* — это ситуация, когда для представления результата в одном из нормальных форматов его необходимо округлить. Термин *invalid* употребляется для описания ситуации, когда предпринимается попытка выполнения недопустимой операции, такой как 0/0 или $\sqrt{-1}$. При возникновении одной из указанных исключительных ситуаций результату присваивается специальное значение.

Если установлен флаг разрешения соответствующего прерывания, в исключительной ситуации происходит переход к системной или пользовательской программе обработки этого прерывания. В качестве альтернативы прикладная программа может сама проверять, нет ли исключений, и производить те или иные операции в соответствии с результатами проверки.

6.7.2. Арифметические операции над числами с плавающей запятой

Этот раздел посвящен правилам сложения, вычитания, умножения и деления чисел в формате с плавающей запятой, определенном стандартом IEEE. Указанные правила регулируют только базовые шаги при выполнении операций. А вот, например, возможность переполнения или потери значимости в них не учтена. Более того, для промежуточных значений мантиссы и порядка может потребоваться более 24 и 8 бит соответственно. Эти и другие особенности операций с плавающей запятой нужно принимать к сведению в процессе разработки арифметических устройств, соответствующих стандарту IEEE. Все особенности мы, конечно, не рассмотрим, но о важнейших, таких как округление, поговорим обязательно.

Если порядок двух операндов с плавающей запятой различен, их мантиссы перед сложением или вычитанием должны быть сдвинуты относительно друг друга. В качестве примера рассмотрим сложение чисел $2{,}9400 \times 10^2$ и $4{,}3100 \times 10^4$. Представим $2{,}9400 \times 10^2$ как $0{,}0294 \times 10^4$ и сложим мантиссы, вследствие чего получим $4{,}3394 \times 10^4$. Последовательность операций при сложении и вычитании можно описать следующим образом.

Правило сложения и вычитания

1. Выбрать число с меньшим порядком и сдвинуть его мантиссу вправо на количество разрядов, равное разности порядков.
2. Установить порядок результата равным большему порядку операндов.
3. Выполнить сложение/вычитание мантисс и определить знак результата.
4. Нормализовать результат в случае необходимости.

Умножение и деление чисел с плавающей запятой даже проще их сложения и вычитания — для выполнения этих операций выравнивать мантиссы не требуется.

Правило умножения

1. Сложить порядки операндов и вычесть из результата значение 127.
2. Перемножить мантиссы и определить знак результата.
3. В случае необходимости нормализовать результат.

Правило деления

1. Вычесть порядок делителя из порядка делимого и прибавить к результату значение 127.
2. Разделить мантиссы и определить знак результата.
3. В случае необходимости нормализовать результат.

Прибавление или вычитание 127 при умножении и делении выполняется потому, что порядки чисел представлены в формате с избытком 127.

6.7.3. Разряды защиты и усечение

Реализация описанных алгоритмов имеет ряд важных особенностей, обзор которых дан в настоящем разделе. Хотя размер мантисс исходных операндов и конечного результата ограничен 24 разрядами, включая подразумеваемую ведущую 1, важно, чтобы в ходе промежуточных вычислений сохранялось несколько дополнительных разрядов, называемых *разрядами защиты* или *сторожевыми разрядами*. Это позволяет обеспечить необходимую точность конечного результата.

Когда разряды защиты удаляются из конечного результата, мантисса усекается до 24 разрядов. Усечение производится и в других ситуациях, в частности, при преобразовании десятичного числа в двоичный формат. Конечно, для обозначения данной операции употребляется также термин *округление*, но мы будем использовать его в более узком смысле, говоря об одном из способов усечения значения.

Существует несколько способов усечения. Простейший из них заключается в удалении разрядов защиты без изменения остальных разрядов. Такая операция называется *усечением*. Предположим, что необходимо сократить дробное значение с шести разрядов до трех. Любые значения, лежащие в диапазоне от $0,b_{-1}b_{-2}b_{-3}000$ до $0,b_{-1}b_{-2}b_{-3}111$, усекаются до $0,b_{-1}b_{-2}b_{-3}$. Ошибка усечения до 3-разрядного результата находится в диапазоне 0–0,000111, то есть от 0 и почти до 1 в младшем из оставшихся разрядов. У нас это разряд b_{-3}. В результате усечения получается *смещенное* приближение, поскольку диапазон ошибки не симметричен нулю.

Еще одним простым методом усечения является *фон-неймановское округление*. Если все удаляемые разряды содержат нули, последние отбрасываются без изменения оставшихся разрядов. Но если хоть один из удаляемых разрядов содержит 1, младший разряд оставшегося значения устанавливается в 1. В нашем примере усечения дробного значения с шести разрядов до трех любые 6-разрядные значения, в которых $b_{-4}b_{-5}b_{-6}$ не равны 000, укорачиваются до $0,b_{-1}b_{-2}1$. Величина ошибки этого метода лежит в диапазоне от −1 до +1 младшего из оставшихся разрядов. И хотя при таком способе усечения значения диапазон ошибки больше, чем при простом усечении, ее максимальная величина остается той же, а приближение получается *несмещенным*, так как диапазон ошибки симметричен относительно нуля.

Для тех операций, в которых участвует много операндов и выполняется значительное количество промежуточных действий, предпочтительнее несмещенное приближение, при котором положительные и отрицательные ошибки компенсируют друг друга. Если статистически оценить результаты сложных вычислений с использованием несмещенного приближения, их точность окажется очень высокой.

Перейдем к следующему методу усечения значения — *округлению*. Задача округления состоит в максимальном приближении результирующего значения к исходному. Оно гораздо точнее полученного путем усечения и к тому же дает несмещенный результат. Округление выполняется так: если старший из удаляемых разрядов содержит 1, к младшему из оставшихся разрядов числа прибавляется 1. Таким образом, $0{,}b_{-1}b_{-2}b_{-3}1...$ округляется до $0{,}b_{-1}b_{-2}b_{-3} + 0{,}001$, а $0{,}b_{-1}b_{-2}b_{-3}0...$ — до $0{,}b_{-1}b_{-2}b_{-3}$. Результат получается предельно близким к исходному числу за исключением ситуации, когда удаляемые разряды равны 10...0. В этом случае исходное значение лежит посредине между двумя возможными усеченными представлениями. Для обеспечения несмещенного приближения можно выбирать значение младшего из оставшихся разрядов таким образом, чтобы всегда получалось ближайшее четное значение. В рассматриваемом примере с использованием 6 разрядов значение $0{,}b_{-1}b_{-2}0100$ округляется до $0{,}b_{-1}b_{-2}0$, а $0{,}b_{-1}b_{-2}1100$ — до $0{,}b_{-1}b_{-2}1 + 0{,}001$. Эту технологию можно описать так: округление до ближайшего числа, а в случае двух одинаковых ошибок округления — до ближайшего четного числа.

Ошибка округления лежит в диапазоне от $-1/2$ до $+1/2$ значения младшего из оставшихся разрядов. Очевидно, что это наилучший метод. Однако реализовать его труднее всего, поскольку потребуется дополнительная операция и, возможно, нормализация. Согласно стандарту IEEE, для усечения чисел по умолчанию используется округление. В этом стандарте описаны и другие методы усечения; все они определены как режимы округления.

Обсуждая ошибки, возникающие из-за удаления разрядов защиты, мы говорили об одной операции усечения. Если программа выполняет длинную последовательность вычислений, в которой задействованы числа с плавающей запятой, анализ диапазонов ошибок и конечных результатов значительно усложняется. Этот аспект числовых вычислений мы больше обсуждать не будем, а напоследок еще раз обратимся к стандарту IEEE и посмотрим, как в ним определяются разряды защиты и округление.

Результаты одной операции должны вычисляться с точностью до половины единицы в младшем разряде. В общем случае с этой целью результат должен укорачиваться путем округления. Для обеспечения такой точности на промежуточных шагах выполнения операции достаточно сохранять три разряда защиты. Первые два — это старшие разряды мантиссы, которые в конце вычислений подлежат удалению. Третий разряд содержит результат выполнения логической операции ИЛИ всех разрядов мантиссы, кроме указанных двух разрядов защиты. Поддерживать его на промежуточных шагах операции достаточно просто. Он инициализируется нулем, а когда в него из мантиссы выдвигается 1, становится равным 1 и сохраняет это значение. Поэтому данный разряд иногда называют вторым промежуточным битом округления (sticky bit).

6.7.4. Операции с плавающей запятой

Для аппаратной реализации операций с плавающей запятой требуются очень сложные схемы. Существует также программный способ реализации. Но в любом случае компьютер должен уметь преобразовывать входные и выходные данные из пользовательского десятичного формата в компьютерный двоичный и наоборот. В большинстве процессоров общего назначения операции с плавающей запятой реализованы аппаратно и доступны на уровне машинных команд.

Пример реализации операций с плавающей запятой приведен на рис. 6.26. Это блок-схема операций сложения и вычитания в формате, использовавшемся при создании рис. 6.24, *а*. Согласно правилу сложения и вычитания, приведенному в разделе 6.7.2, первым шагом является сравнение порядков, производимое с целью определить, насколько следует сдвинуть мантиссу числа с меньшим порядком. Значение счетчика сдвига n определяется 8-разрядной схемой вычитателя (левый верхний угол рисунка). Разность $E'_A - E'_B$, то есть значение n, направляется на вход устройства сдвига. Если n больше количества значащих разрядов операнда, результатом операции будет больший операнд (если не учитывать разряды защиты, участвующие в округлении). Этот случай мы не обсуждаем.

Знак разности, полученной вследствие сравнения порядков, определяет, какая из мантисс подлежит сдвигу. Поэтому на первом шаге знак подается на вход схемы SWAP (правый верхний угол рисунка). Если знак равен 0, это означает, что $E'_A \geq E'_B$, и мантиссы M_A и M_B проходят через схему SWAP без изменений. В результате M_B направляется в блок сдвига, где она сдвигается вправо на n разрядов. Вторая мантисса, M_A, направляется непосредственно на сумматор/вычитатель мантисс. Если же знак разности равен 1, то $E'_A < E'_B$ и перед отправкой в блок сдвига блок SWAP поменяет мантиссы местами.

Шаг 2 выполняется двухвходовым мультиплексором. На основе знака разности, полученного при сравнении порядков на шаге 1, порядок результата, E', определяется как E'_A, если $E'_A \geq E'_B$, или как E'_B, если $E'_A < E'_B$.

Для выполнения шага 3 используется главный компонент схемы, сумматор/вычитатель мантисс, изображенный в средней части рисунка. Логическая схема управления определяет, какую операцию производить с мантиссами — сложение или вычитание. Решение зависит от знака операнда (S_A и S_B) и операции (сложение или вычитание), которая должна быть выполнена по отношению к операндам. Этот же блок определяет знак результата, S_R. Например, если A отрицательное число ($S_A = 1$), B — положительное ($S_B = 0$) и выполняется операция $A - B$, то мантиссы складываются и результат также будет отрицательным ($S_R = 1$). Когда A и B положительные, при выполнении операции $A - B$ осуществляется вычитание мантисс. Знак результата, S_R, в этом случае зависит от операции вычитания мантисс. Например, если порядок $E'_A > E'_B$, значит, разность M_A – (сдвинутая M_B) положительна, следовательно, и результат положительный. Если же $E'_B > E'_A$, положительной будет разность M_B – (сдвинутая M_A), а результат операции получит отрицательный знак. Этот пример демонстрирует, что знак результата сравнения порядков тоже должен подаваться на вход схемы управления. Когда $E'_A = E'_B$, при вычитании мантисс знак результата определяется знаком на выходе сумматора/вычитателя мантисс. Теперь читатель может самостоятельно составить полную таблицу истинности схемы управления.

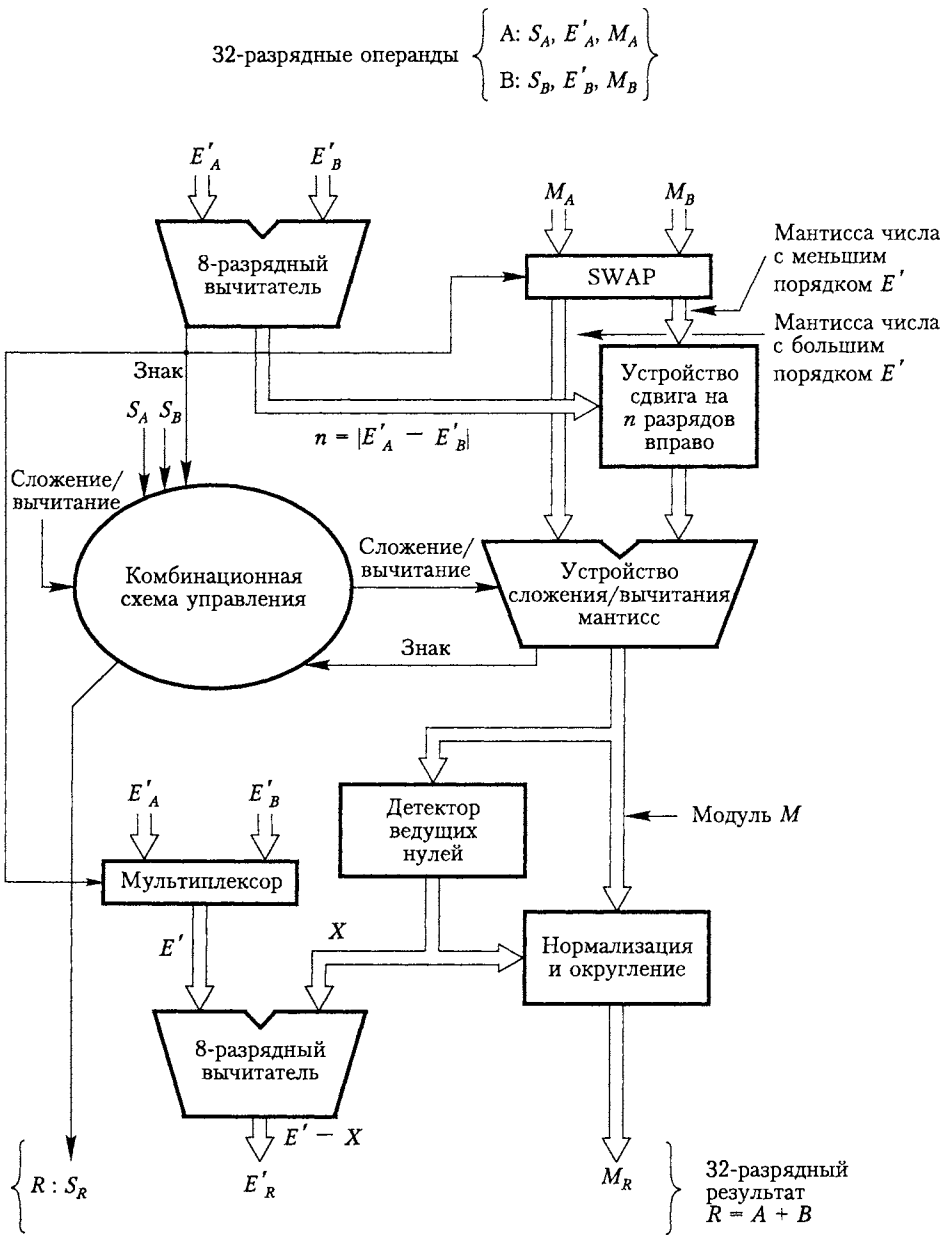


Рис. 6.26. Устройство сложения/вычитания чисел с плавающей запятой

На четвертом шаге сложения/вычитания выполняется нормализация результата третьего шага, то есть мантиссы M. Количество ведущих нулей в M определяет число разрядов, X, на которое нужно сдвинуть M. Нормализованная таким образом мантисса результата усекается до 24 разрядов, вследствие чего получаем

окончательное значение мантиссы — M_R. Для определения окончательного значения порядка E'_R из промежуточного значения порядка E' нужно вычесть значение X. Следует отметить, что для нормализации результата может оказаться достаточно одного сдвига вправо. Так производится и сложение двух мантисс в форме 1,*xx*.... В этом случае вектор M имеет форму 1*x*,*xx*..., что на рисунке соответствует значению X, равному −1.

Мы не обсуждали принципы обработки разрядов защиты промежуточных значений мантиссы. Выше лишь было отмечено, что, согласно стандарту IEEE, для формирования 24-разрядной нормализованной мантиссы результата достаточно всего нескольких разрядов защиты.

Какие же аппаратные компоненты нужны для реализации логических блоков, показанных на рис. 6.26? Два 8-разрядных вычитателя и сумматор/вычитатель мантисс можно реализовать на основе комбинационной логики. Поскольку они должны генерировать выходные данные в формате значения со знаком, в описанную ранее структуру этих схем нужно внести некоторые поправки. В подобных схемах часто комбинируют арифметику дополнений до единицы и представление значения со знаком. Для реализации блоков сдвига и нормализации результата используются разные схемные решения. Если нужна структура с современным вентильным счетчиком, можно применять сдвиговые регистры. Для повышения производительности их следует реализовать в виде комбинационных логических схем, что, однако, потребует большого числа логических вентилей. В высокопроизводительных процессорах значительная часть площади микросхемы выделяется для выполнения операций с плавающей запятой.

6.8. Резюме

С компьютерной арифметикой связано несколько интересных архитектурных задач, касающихся структуры логических схем. В этой главе рассмотрен ряд технологий, широко используемых при разработке двоичных арифметических устройств. Одним из ключевых принципов построения высокопроизводительных сумматоров является принцип параллельного переноса. Для ускорения работы умножителей применяется операция перекодировки пар разрядов, которую можно считать усовершенствованным вариантом алгоритма Бута. Он позволяет сократить количество слагаемых, необходимых для получения произведения. Технология сложения с сохранением переноса значительно ускоряет операцию.

Значительное внимание уделялось стандарту IEEE, который определяет способы представления чисел с плавающей запятой и набор правил выполнения с такими числами четырех стандартных арифметических операций. Чтобы показать, насколько сложны логические схемы, выполняющие операции с плавающей запятой, мы привели пример блок-схемы устройства сложения/вычитания.

Упражнения

6.1. В следующих задачах на сложение и вычитание операндами являются 6-разрядные двоичные числа со знаком в форме дополнения до двух. Выполните указанные операции и для каждой из них укажите, произошло ли арифметическое переполнение, а потом проверьте свои ответы, преобразовав операнды и результаты в десятичные числа формата значения со знаком.

```
 010110   101011   111111
+001001  +100101  +000111

 011001   110111   010101
+010000  +111001  +101011

 010110   111110   100001
-011111  -100101  -011101

 111111   000111   011010
-000111  -111000  -100010
```

6.2. В начале раздела 6.7 рассказывалось о представлении двоичных дробей со знаком в формате дополнения до двух.

а) Представьте десятичные числа 0,5, −0,123, −0,75 и −0,1 в виде 6-разрядной дроби со знаком (см. приложение Д, где описывается преобразование дробей из десятичного формата в двоичный).

б) Какова максимальная ошибка представления, e, при использовании только 5 значащих разрядов после двоичной запятой?

в) Вычислите количество разрядов после двоичной запятой, необходимое для выполнения следующих условий:

$$1)\ e < \frac{1}{10};$$

$$2)\ e < \frac{1}{100};$$

$$3)\ e < \frac{1}{1000};$$

$$4)\ e < \frac{1}{10^6}.$$

6.3 Двоичные представления в формате дополнения до единицы и до двух являются частными случаями представления чисел в системе счисления с основанием b в формате дополнения до $(b-1)$ и до b. Для примера возьмем десятичную систему. Числа в формате значения со знаком +526, −526, +70 и −70 можно представить в каждой из двух систем дополнения в виде чисел со знаком из четырех цифр, как показано на рис. У6.1. Значение числа в системе дополнения до девяти формируется путем дополнения каждой цифры до 9. А для получения дополнения до десяти к дополнению до 9

прибавляется 1. В каждом из этих двух представлений крайняя слева цифра положительных чисел — 0, а крайняя слева цифра отрицательных чисел — 9.

Представление	Примеры			
Значение со знаком	+526	−526	+70	−70
Дополнение до 9	0526	9473	0070	9929
Дополнение до 10	0526	9474	0070	9930

Рис. У6.1. Числа со знаком по основанию 10 в упражнении 6.3

Рассмотрим систему счисления с основанием 3 (троичную систему), в которой пятизначное число со знаком $t_4t_3t_2t_1t_0$ имеет значение $t_4 \times 3^4 + t_3 \times 3^3 + t_2 \times 3^2 + t_1 \times 3^1 + t_0 \times 3^0$, где $0 \le t_i \le 2$. Представьте троичные числа в формате значения со знаком +11011, −10222, +2120, −1212, +10 и −201 в виде 6-значных троичных чисел со знаком в системе дополнения до трех.

6.4. Представьте десятичные числа 56, −37, 122 и −123 в виде 6-значных чисел со знаком в троичном формате дополнения до трех. Выполните сложение и вычитание для всех возможных пар этих чисел. Для каждой операции укажите, произошло ли арифметическое переполнение. (Определение троичной системы дано в упражнении 6.3. Для преобразования чисел из десятичной системы в троичную используйте процедуру, аналогичную описанной в приложении Д.)

6.5. Полусумматор — это комбинационная логическая схема с двумя входами, x и y, и двумя выходами, c и s, представляющими сумму и перенос, полученные в результате двоичного сложения x и y.

а) Разработайте полусумматор в виде двухуровневой схемы И-ИЛИ.

б) Покажите, как реализовать полный сумматор, показанный на рис. 6.2, *а*, на основе двух полусумматоров и внешних логических вентилей.

в) Сравните задержку на самом длинном пути через схему, которую вы разработали в упражнении 6.5, *б*, с задержкой в схеме сумматора, приведенной на рис. 6.2, *а*.

6.6. Напишите программу для процессора 68000 или IA-32, преобразующую 16-разрядные положительные двоичные числа в 5-значные десятичные числа. Каждая цифра должна быть представлена в двоично-десятичном коде (BCD). BCD-коды цифр занимают 4 младших разряда последовательных байтов в основной памяти компьютера. Для преобразования используйте технологию последовательного деления на 10. Этот метод аналогичен последовательному делению на 2 для преобразования из десятичного формата в двоичный (см. приложение Д). Формат и действие команды Divide описаны в приложениях В (процессор 68000) и Г (процессор IA-32).

6.7. Предположим, что четыре цифры в формате BCD, представляющие десятичное целое число из диапазона от 0 до 9999, хранятся в младшей половине

32-разрядного слова памяти по адресу DECIMAL. Напишите подпрограмму для процессора ARM, 68000 или IA-32, преобразующую десятичные целые числа, хранящиеся по адресу DECIMAL, в двоичное представление и записывающую их в память по адресу BINARY.

6.8. Для сложения цифр BCD нужен сумматор по модулю 10. Сложение двух цифр BCD по модулю 10, $A = A_3A_2A_1A_0$ и $B = B_3B_2B_1B_0$, можно выполнить так. Сначала прибавляем A к B (двоичное сложение). Затем, если результатом окажется неверный код, который больше или равен 10_{10}, прибавляем 6_{10}. (Переполнение при сложении игнорируем.)

а) В каком случае выходной перенос равен 1?

б) Докажите, что алгоритм позволяет получить правильные результаты в следующих случаях:

1) $A = 0101$ и $B = 0110$

2) $A = 0011$ и $B = 0100$

в) Разработайте сумматор для цифр BCD на основе 4-разрядного двоичного сумматора и внешних логических вентилей. На его входы подаются сигналы $A_3A_2A_1A_0$, $B_3B_2B_1B_0$ и входной перенос. На выходах должны быть цифра суммы $S_3S_2S_1S_0$ и выходной перенос. Каскад таких блоков может образовать BCD-сумматор с последовательным переносом.

6.9. С помощью таблицы истинности докажите, что такое логическое выражение, как $c_n \oplus c_{n-1}$ можно использовать в качестве индикатора переполнения при сложении целых чисел в системе дополнения до двух.

6.10. а) Разработайте 64-разрядный сумматор, который состоит из четырех 16-разрядных сумматоров с параллельным переносом и дополнительной логики, генерирующей c_{16}, c_{32}, c_{48} и c_{64} на основе значений c_0, G_i^{II} и P_i^{II} (см. рис. 6.5). Какова связь между дополнительной логикой и логикой внутри каждой схемы с параллельным переносом?

б) Докажите, что в 64-разрядном сумматоре для формирования суммы s_{63} требуется 12 вентильных задержек, а для формирования суммы c_{64} нужно 7 вентильных задержек, как утверждалось в разделе 6.2.1.

в) Сравните количество вентильных задержек для s_{63} и c_{32} в 64-разрядном сумматоре из упражнения 6.10, *а* с количеством вентильных задержек для тех же переменных в 32-разрядном сумматоре в виде каскада из двух 16-разрядных сумматоров, который был описан в разделе 6.2.1.

6.11. а) Сколько логических вентилей требуется для создания 4-разрядного сумматора с параллельным переносом, показанного на рис. 6.4?

б) Используя некоторые из результатов вычислений, выполненных в упражнении 6.11, *а*, определите количество логических вентилей, необходимых для создания 16-разрядного сумматора с параллельным переносом, показанного на рис. 6.5.

6.12. Докажите, что в матрице $n \times n$, имеющей тот же тип, что и у приведенной на рис. 6.6, *б*, задержка в худшем случае равна $6(n - 1) - 1$ вентильным задержкам, как утверждалось в разделе 6.3.

6.13. Выполните вручную операции $A \times B$ и $A \div B$ над 5-разрядными числами без знака A =10101 и B = 00101.

6.14. Продемонстрируйте, как операции умножения и деления из упражнения 6.13 выполняются схемами, показанными на рис. 6.7, а и 6.21. Для этого составьте такие схемы, как на рис. 6.7, б и 6.23.

6.15. Напишите программу для процессора ARM, 68000 или IA-32, которая выполняла бы умножение двух 32-разрядных чисел без знака по технологии, показанной на рис. 6.7. Предполагается, что множитель и множимое находятся в регистрах R_2 и R_3. Произведение помещается в регистры R_1 (старшие разряды) и R_2 (младшие разряды). (Подсказка: для сдвига в двух регистрах используйте операции сдвига и циклического сдвига.)

6.16. Напишите программу для процессора ARM, 68000 или IA-32, осуществляющую целочисленное деление на основе алгоритма деления без восстановления. Предполагается, что оба операнда положительны, то есть крайний слева разряд каждого из них равен нулю.

6.17. Перемножьте числа со знаком в формате дополнения до двух с использованием алгоритма Бута. Предполагается, что A — множимое, а B — множитель.

1) A = 010111 и B = 110110
2) A = 110011 и B = 101100
3) A = 110101 и B = 011011
4) A = 001111 и B = 001111

6.18. Повторите упражнение 6.17, используя алгоритм перекодировки пар разрядов множителей.

6.19. Покажите, каким образом следует модифицировать схему, представленную на рис. 6.7, *а*, для умножения n-разрядных чисел со знаком в формате дополнения до двух, если используется алгоритм Бута. Явно определите входы и выходы управляющего секвенсора. Укажите, какие изменения связаны с сумматором и регистром A.

6.20. Если произведение двух n-разрядных чисел со знаком в формате дополнения до двух можно представить с помощью n разрядов, то при умножении этих чисел допускается непосредственное использование алгоритма ручного умножения (рис. 6.6, *а*). При этом знаковый разряд обрабатывается так же, как остальные разряды. Правильно ли это по отношению к следующим парам 4-разрядных чисел со знаком:

1) Множимое = 1110 и Множитель = 1101
2) Множимое = 0010 и Множитель = 1110

Почему этот алгоритм работает корректно?

6.21. Целочисленное арифметическое устройство, которое может выполнять сложение и умножение 16-разрядных чисел без знака, должно использоваться для умножения двух 32-разрядных чисел без знака. Все операнды, промежуточные и конечные результаты хранятся в 16-разрядных регистрах от R_0 до R_{15}. Умножитель, реализованный аппаратно, перемножает содержимое регистров R_i (множимое) и R_j (множитель) и помещает 32-разрядное произведение двойной длины в регистры R_j (младшие разряды) и R_{j+1}. При $j = i - 1$ произведение замещает оба операнда. Аппаратно реализованный сумматор складывает содержимое регистров R_i и R_j и помещает результат в R_j. Входной перенос операции сложения равен 0, а входной перенос операции сложения с переносом является значением флага переноса C. Выходной перенос из сумматора всегда хранится в C.

Опишите шаги процедуры умножения двух 32-разрядных операндов, хранящихся в парах регистров R_1, R_0 и R_3, R_2 (первый регистр каждой пары содержит старшие разряды, а второй — младшие). Запишите 64-разрядное произведение в регистры R_{15}, R_{14}, R_{13} и R_{12}. В любом из регистров от R_{11} до R_4 при необходимости можно хранить промежуточные значения, а на каждом шаге процедуры может выполняться умножение, сложение или пересылка значений из регистра в регистр.

6.22. а) Вычислите количество вентильных задержек при формировании разряда произведения p_7 в каждой из матриц, показанных на рис. 6.16. Предполагается, что все выходные значения полного сумматора выдаются через две вентильные задержки после подачи входных значений. Учтите задержку в вентиле И во время формирования произведения $m_i q_j$ в начале процесса умножения.

б) В разделе 6.4 утверждалось, что задержка в схеме на рис. 6.16, а, расширенной для умножения $n \times n$ разрядов, равна $6(n - 1) - 1$. Выведите аналогичное выражение для расширения схемы, предназначенной для умножения $n \times n$ разрядов (рис. 6.16, *б*).

6.23. Как получить формулу, по которой вычисляется количество шагов сложения с сохранением переноса, необходимое для сведения k слагаемых к двум векторам. (Эта формула была приведена без доказательства в разделе 6.5.2.)

6.24. а) Сколько уровней CSA необходимо для сокращения 16 слагаемых до двух, если использовать схему, подобную приведенной на рис. 6.19?

б) Нарисуйте схему сокращения 32 слагаемых до двух, подтверждающую, что для этого потребуется 8 уровней CSA, как это было сказано в разделе 6.5.2.

в) Сравните точные ответы к упражнениям 6.24, *а* и 6.24, *б* с приблизительными результатами, полученными вследствие вычислений по формуле $1{,}7 \log_2 k - 1{,}7$.

6.25. Работая над разделом 6.7, для представления чисел с плавающей запятой мы использовали реальный 32-разрядный формат стандарта IEEE. В данном упражнении мы будем применять укороченный формат (см. рис. У6.2.)

Масштабный множитель имеет подразумеваемое основание 2 и 5-разрядный показатель степени в формате с избытком 15, с двумя граничными значениями, 0 и 31, необходимыми для обозначения точного 0 и бесконечности. Предполагается, что 6-разрядная мантисса нормализована так же, как в формате IEEE, с подразумеваемой единицей слева от двоичной запятой.

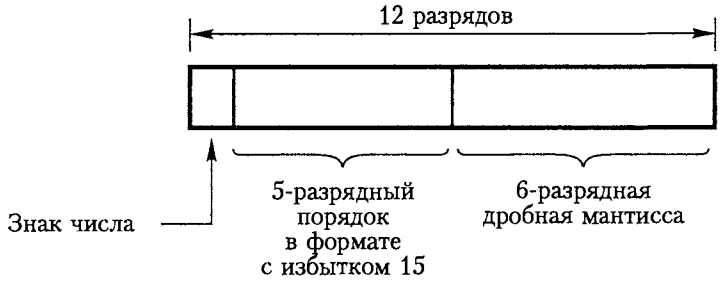


Рис. У6.2. Формат числа с плавающей запятой, используемый в упражнении 6.25

а) Представьте в этом формате числа: + 1,7, −0,012, +19 и 1/8.

б) Каковы наименьшее и наибольшее числа, которые можно представить в данном формате?

в) Сравните диапазон, вычисленный в упражнении 6.25, *а*, с диапазонами для 12-разрядного целого числа со знаком и 12-разрядной дроби со знаком?

г) Выполните над следующими операндами операции сложения, вычитания, умножения и деления:

A =	0	10001	011011
B =	1	01111	101010

6.26. Рассмотрим 16-разрядное число с плавающей запятой в формате, подобном описанному в упражнении 6.25, с 6-разрядным порядком и 9-разрядной нормализованной дробной мантиссой. Основание масштабного множителя равно 2, а порядок представлен в формате с избытком 31.

а) Сложите приведенные ниже числа A и B:

A =	0	100001	111111110
B =	0	011111	001010101

Представьте ответ в нормализованной форме. Помните, что слева от двоичной запятой располагается подразумеваемая единица, не включенная в форматы чисел A и B. Для получения результирующей нормализованной 9-разрядной мантиссы выполните округление.

б) Используя десятичные числа w, x, y и z, выразите наибольшее и наименьшее (ненулевое) значения, которые можно представить в описанном выше нормализованном формате с плавающей запятой. Форма ответа должна быть такой:

$$\text{Наибольшее} = w \times 2^{x}$$
$$\text{Наименьшее} = y \times 2^{-z}$$

6.27. Как представление с избытком x порядка числа с плавающей запятой в формате, приведенном на рис. 6.24, *а*, упрощает сравнение относительного размера двух чисел с плавающей запятой? (Подсказка: предположим, имеется комбинационная логическая схема, сравнивающая относительные размеры двух 32-разрядных целых чисел без знака. Используйте эту схему и дополнительные логические вентили для разработки схемы, сравнивающей числа с плавающей запятой.)

6.28. В упражнении 6.25, *а* применялся способ преобразования простых десятичных чисел в двоичный формат с плавающей запятой. Однако если десятичные числа заданы в формате с плавающей запятой, процесс преобразования более сложен, поскольку нельзя по отдельности конвертировать мантиссу и показатель степени масштабного множителя. Ведь в общем случае в уравнении $10^{x} = 2^{y}$ оба числа, x и y, не могут быть целыми. Предположим, имеется таблица двоичных чисел с плавающей запятой t_i, где $t_i = 10^{x_i}$ для x_i в представимом диапазоне. Опишите процедуру преобразования заданного десятичного числа с плавающей запятой в число двоичного формата с плавающей запятой. Используйте любые поддерживаемые компьютером команды — как целочисленные, так и с плавающей запятой.

6.29. Представьте десятичное число 0,1 в виде 8-разрядной двоичной дроби в формате, описанном в разделе 6.7. Если число нельзя преобразовать в 8-разрядный формат без потери точности, определите его приближенные значения, используя три метода усечения, описанные в разделе 6.7.3.

6.30. Приведите пример, демонстрирующий, что для получения правильного результата вычитания двух положительных чисел нужно три разряда защиты.

6.31. Какой из четырех 6-разрядных результатов, полученных при решении задачи 6.2, *а*, неточен? Для каждого из случаев приведите три 6-разрядных значения, которые соответствуют трем типам усечения, описанным в разделе 6.7.3.

6.32. Выведите логические формулы, определяющие выходы сложения/вычитания и S_R комбинационной схемы управления на рис. 6.26.

6.33. Если коэффициент объединения по входу ограничен четырьмя вентилями, какой должна быть комбинационная реализация блока сдвига на рис. 6.26?

6.34. а) Нарисуйте логическую вентильную схему, реализующую мультиплексор, показанный на рис. 6.26.

б) Как структура схемы SWAP (рис. 6.26) связана с вашим решением упражнения 6.34, *а*?

6.35. Какова логическая реализация детектора ведущих нулей на рис. 6.26?

6.36. Сумматор-вычитатель мантисс на рис. 6.26 оперирует положительными беззнаковыми двоичными дробями и генерирует результат в виде значения со знаком. Обсуждая рис. 6.26, мы утверждали, что для данного формата входных и выходных операндов удобна арифметика дополнения до единицы. При сложении двух чисел со знаком в формате дополнения до единицы для получения правильного ответа к результату должен прибавляться перенос из знакового разряда. Эта операция называется *поправкой путем циклического переноса*. Рассмотрим два примера сложения (рис. У6.3), в которых операнды и ответ представлены в форме 4-разрядных чисел со знаком в системе дополнения до единицы.

Система дополнения до единицы удобна в случаях, когда результат требуется сгенерировать в формате значения со знаком, так как для преобразования числа, представленного в формате дополнения до единицы, в формат значения со знаком достаточно заменить разряды, находящиеся справа от знакового разряда, их дополнениями. Если использовать арифметику дополнений до двух, для преобразования отрицательного числа в формат значения со знаком требуется выполнить прибавление единицы. В случае применения сумматора с параллельным переносом можно включить операцию циклического переноса, выполняемую при сложении чисел со знаком в формате дополнения до единицы, в логику параллельного переноса. Создайте полную схему сумматора-вычитателя для схемы, приведенной на рис. 6.26.

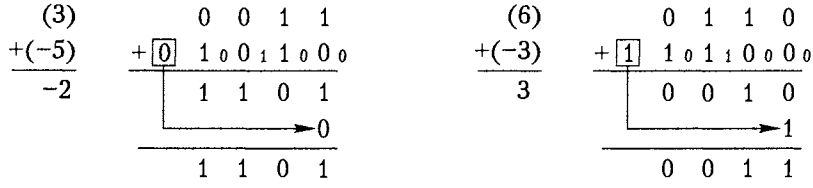


Рис. У6.3. Сложение в упражнении 6.36

Глава 7

Процессор

- Как процессор выполняет команды
- Внутренние функциональные блоки процессора и их взаимодействие
- Аппаратное обеспечение, генерирующее внутренние управляющие сигналы
- Микропрограммная организация процессорного устройства

Настоящая и несколько следующих глав посвящены описанию процессорного устройства, которое выполняет машинные команды и координирует действия других устройств. Его часто называют *процессором системы команд* (Instruction Set Processor, ISP) или просто *процессором*. Мы проанализируем внутреннюю структуру процессора и расскажем о том, как он осуществляет выборку, производит дешифрацию и выполняет команды программы. Процессорное устройство компьютера обычно называют *центральным процессором* (Central Processing Unit, CPU). Термин «центральный» в свое время отвечал реальному положению дел, поскольку в компьютере был только один процессор команд, но теперь во многих современных компьютерных системах имеется по нескольку процессоров, а следовательно, данный термин явно устарел.

Внутренняя организация процессоров постоянно совершенствуется, отражая развитие технологий и потребность рынка во все более производительных устройствах. Общая стратегия создания высокопроизводительных процессоров направлена на обеспечение параллельной работы как можно большего количества различных функциональных устройств. В частности, такие процессоры имеют конвейерную организацию, при которой выполнение очередной команды начинается до завершения предыдущей. При другом подходе, называемом суперскалярным функционированием, из памяти выбираются и одновременно выполняются несколько команд. Более подробно о конвейерной и суперскалярной архитектуре будет рассказано в главе 8. В настоящей же главе мы сконцентрируем внимание на основных идеях, общих для всех процессоров.

Типичная компьютерная задача состоит из цепочки шагов, определяемых последовательностью машинных команд программы. Каждая команда разбивается процессором на ряд элементарных машинных операций. Эти операции и принципы управления их выполнением и являются темой данной главы.

7.1. Базовые концепции

Для выполнения программы процессор по одной выбирает команды из памяти и выполняет определяемые ими действия. Команды выбираются из последовательных адресов памяти, пока не встретится команда перехода или ветвления. Для этого в счетчике команд, PC, отслеживается адрес очередной подлежащей выполнению команды. После выборки этой команды содержимое регистра PC обновляется, чтобы он указывал на следующую команду в памяти в порядке расположения адресов. Команда ветвления может загрузить в PC другой адрес.

Еще одним важнейшим регистром процессора, связанным с выполнением команд, является регистр команды, IR. Предположим, что каждая команда имеет длину 4 байта и хранится в одном слове памяти. Для ее выполнения процессор должен произвести следующие шаги.

1. Выбрать из памяти слово, на которое указывает PC. Содержимое этого слова интерпретируется как команда и загружается в регистр IR. Символически это можно записать так:

$$IR \leftarrow [[PC]]$$

2. Если память адресуется побайтово, следует увеличить содержимое регистра PC на 4:

$$PC \leftarrow [PC] + 4$$

3. Выполнить действия, определяемые командой, которая находится в IR.

Если команда занимает более одного слова, шаги 1 и 2 повторяются столько раз, сколько нужно для выборки всей команды. Эти два шага обычно называют *фазой выборки*, а шаг 3 составляет *фазу выполнения*.

Для детального изучения указанных операций нам прежде всего нужно проанализировать внутреннюю структуру процессора. Главные его блоки представлены на рис. 1.2. Их организация и связи между ними, как вы помните, могут быть разными. Мы начнем с самой простой организации. Позже в этой главе, а также в главе 8 вы познакомитесь с более сложными структурами, предназначенными для обеспечения высокой производительности. На рис. 7.1 показана архитектура процессора, при которой арифметико-логическое устройство (АЛУ) и все регистры соединены одной общей шиной. Это внутренняя шина процессора, которую не следует путать с внешней шиной, соединяющей процессор с основной памятью и устройствами ввода-вывода.

Линии данных и адреса внешней шины памяти на рис. 7.1 соединены с внутренней шиной процессора через регистр данных памяти, MDR, и регистр адреса памяти, MAR. У регистра MDR имеется два входа и два выхода. Данные могут загружаться в него либо с внешней шины памяти, либо с внутренней шины процессора. Хранящиеся в MDR данные также могут быть помещены на любую из этих шин. Вход регистра MAR соединен с внутренней шиной, а его выход — с внешней. Управляющие линии шины памяти соединены с дешифратором команды

и управляющим логическим блоком. Это устройство отвечает за выдачу сигналов, которые управляют работой всех устройств внутри процессора и взаимодействием с шиной памяти.

Количество регистров процессора с именами от R0 до R(n–1) в различных процессорах может быть совершенно разным. Это регистры общего назначения, используемые программистами для нужд программ. Некоторые из них могут быть выделены как регистры специального назначения, например как индексные регистры или указатели стека. Три показанных на рис. 7.1 регистра — Y, Z и TEMP — мы еще не упоминали. Эти регистры прозрачны для программиста — о них не нужно думать, поскольку в командах они никогда явно не указываются и используются процессором для временного хранения информации в ходе выполнения некоторых команд. Регистры Y, Z и TEMP не предназначены для хранения данных, сгенерированных одной командой, для последующего применения другой командой. На вход A арифметико-логического устройства мультиплексор MUX подает либо выходной сигнал регистра Y, либо константу 4. Константа 4, как вы понимаете, увеличивает содержимое счетчика команд. Два возможных значения управляющего входа мультиплексора, определяющих выбор константы 4 или регистра Y, мы будем обозначать как Select4 и SelectY.

В ходе выполнения команды данные пересылаются из одного регистра в другой и в процессе обработки часто попадают в АЛУ, где над ними выполняются арифметические или логические операции. Дешифратор команды и управляющий логический блок отвечают за определение и выполнение действий, заданных командой, которая загружена в регистр IR. Дешифратор генерирует управляющие сигналы, необходимые для выбора регистров, участвующих в выполнении заданной команды, и управляет пересылкой данных. Регистры, АЛУ и внутренняя шина процессора вместе взятые составляют *тракт данных* (datapath).

Процесс выполнения команды — это не что иное, за малым исключением, как реализация в определенной последовательности одной или нескольких из перечисленных ниже операций:

- пересылка слова данных из одного регистра процессора в другой регистр или в АЛУ;
- выполнение арифметической или логической операции и сохранение результата в регистре процессора;
- выборка содержимого заданного адреса памяти и загрузка его в регистр процессора;
- сохранение слова данных из регистра процессора по заданному адресу основной памяти.

Мы подробно рассмотрим процесс выполнения каждой из этих операций, используя модель процессора, приведенную на рис. 7.1.

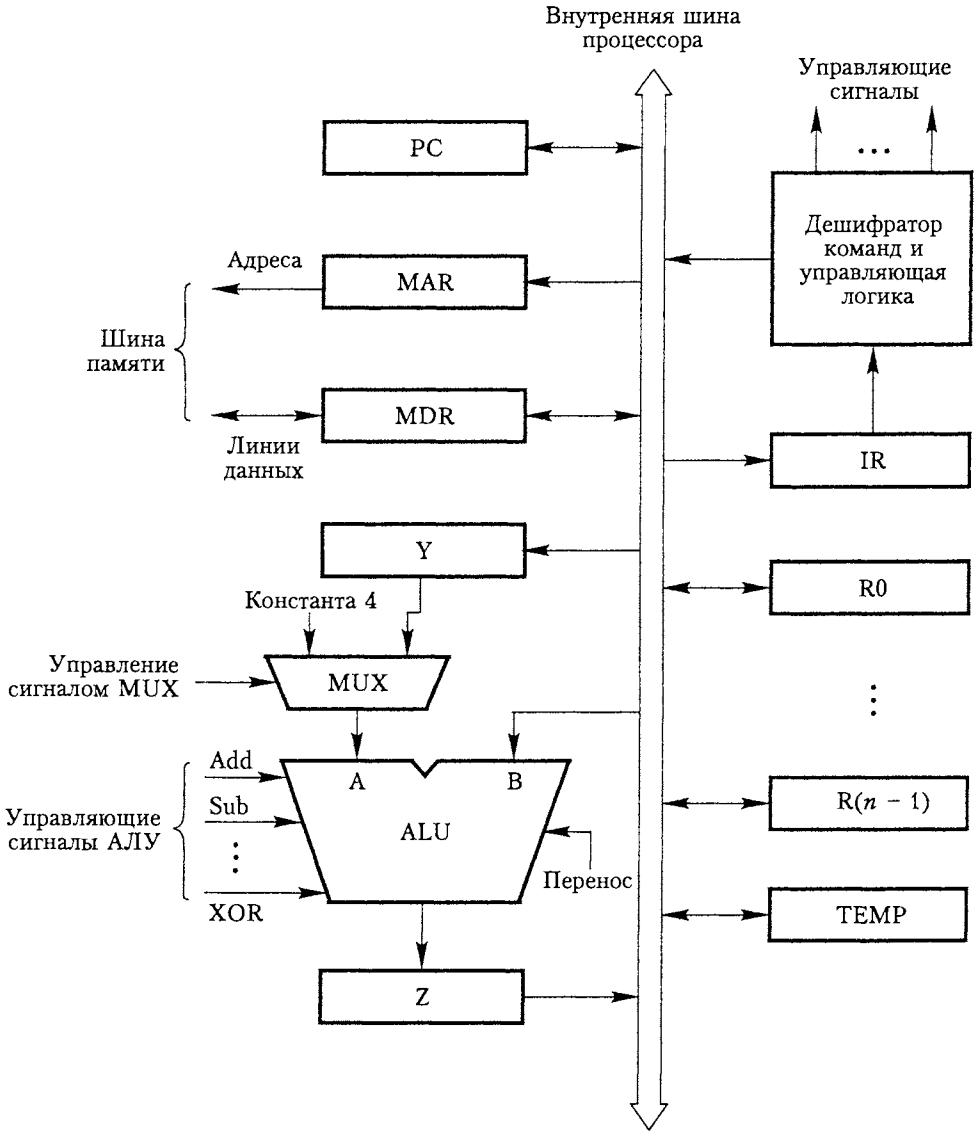


Рис. 7.1. Организация тракта данных внутри процессора с единственной шиной

7.1.1. Пересылка данных между регистрами

В ходе выполнения команд данные постоянно пересылаются из одного регистра в другой. За помещение содержимого регистра на шину и загрузку данных с шины в регистр отвечают два сигнала, символически показанных на рис. 7.2. Вход и выход регистра R*i* соединяются с шиной через ключи, управляемые сигналами Ri_{in} и Ri_{out}. Когда Ri_{in} устанавливается в 1, находящиеся на шине данные загружаются

в регистр R*i*. Аналогичным образом, когда Ri_{out} устанавливается в 1, данные из регистра R*i* помещаются на шину. Но если $Ri_{out} = 0$, шина может использоваться для пересылки данных других регистров.

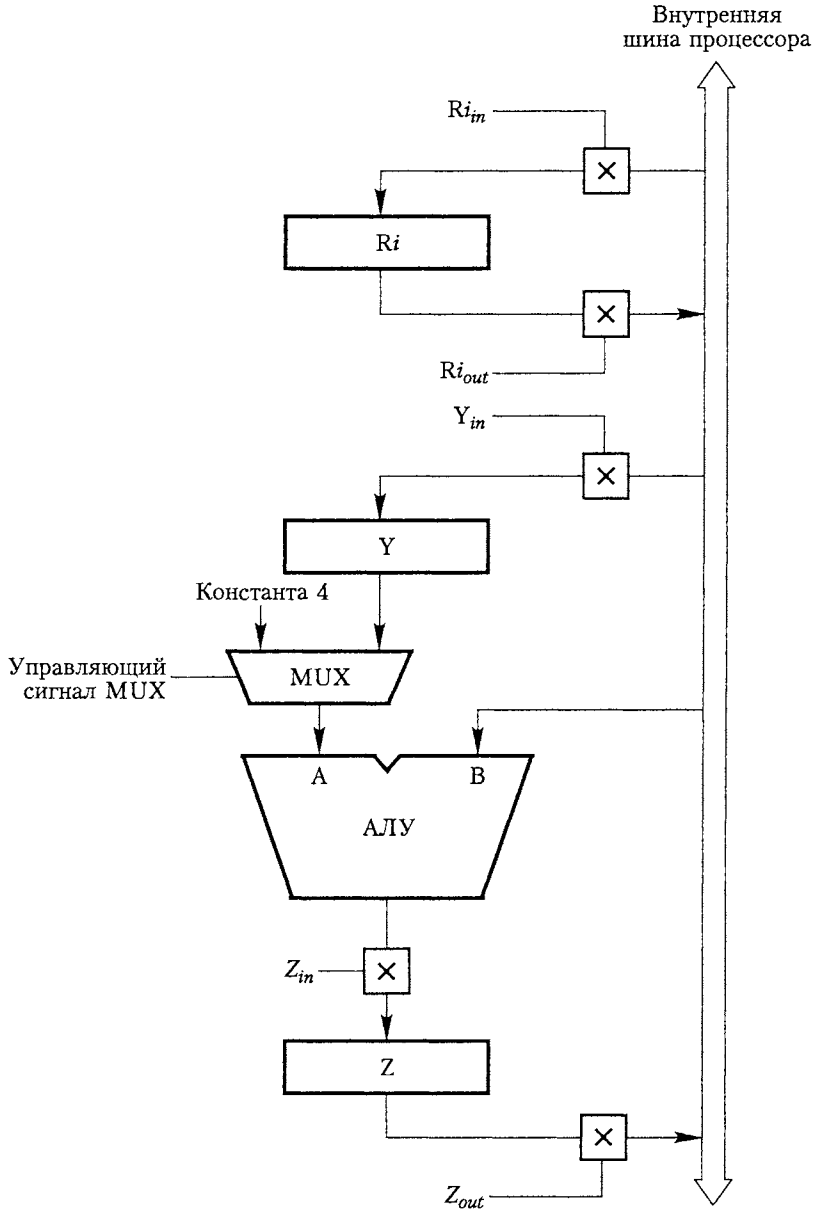


Рис. 7.2. Входные и выходные вентили регистров, показанных на рис. 7.1

Предположим, что мы хотим переслать содержимое регистра R1 в регистр R4. Это можно сделать в два этапа:

- активизируем выход регистра R1, установив $R1_{out}$ в 1, в результате чего содержимое R1 будет помещено на шину процессора;
- активизируем вход регистра R4, установив $R4_{in}$ в 1, и данные с шины процессора будут загружены в регистр R4.

Все операции по пересылке данных внутри процессора выполняются в течение периодов времени, определяемых *тактовым сигналом процессора*. Сигналы, управляющие конкретными операциями пересылки, активизируются в начале такта. В нашем примере $R1_{out}$ и $R4_{in}$ устанавливаются в 1. Регистры состоят из триггеров, управляемых фронтом сигнала. Поэтому на следующем активном фронте сигнала триггеры, составляющие регистр R4, загрузят данные, переданные на их входы. Одновременно с этим управляющие сигналы $R1_{out}$ и $R4_{in}$ опять будут установлены в 0. Эту простую модель тактирования процесса пересылки данных мы будем применять до конца главы. Однако возможны и другие схемы пересылки данных, например, с использованием для этой цели обоих фронтов сигнала. Кроме того, в тех случаях, когда в процессоре не задействуются триггеры, тактируемые фронтом сигнала, для обеспечения корректной пересылки данных могут быть использованы два или более тактовых сигналов. Такое тактирование называется *многофазным*.

Схема реализации одного разряда регистра R*i* показана на рис. 7.3. Для выбора данных, подаваемых на вход тактируемого фронтом сигнала D-триггера, используется двухвходовый мультиплексор. Когда значение сигнала на управляющем входе Ri_{in} равно 1, мультиплексор считывает данные шины. Эти данные будут загружены в триггер по переднему фронту сигнала. Когда Ri_{in} равен 0, мультиплексор помещает на шину текущие данные триггера.

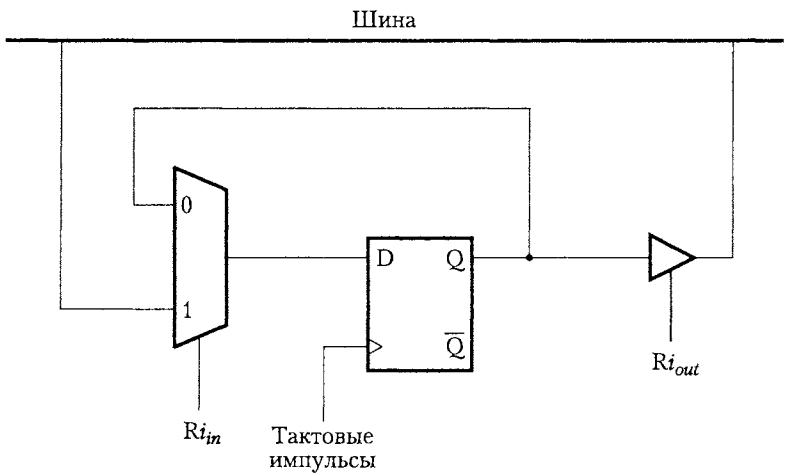


Рис. 7.3. Входные и выходные вентили одного разряда регистра

Выход триггера Q соединяется с шиной через вентиль, имеющий три состояния. Когда Ri_{out} равен 0, выход вентиля находится в высокоимпедансном (электрически отсоединенном) состоянии, которое соответствует открытому ключу. Когда Ri_{out} равен 1, вентиль передает на шину 0 или 1, что зависит от значения Q.

7.1.2. Выполнение арифметической или логической операции

АЛУ представляет собой комбинационную схему, то есть такую, которая не способна хранить данные. Это устройство выполняет арифметические и логические операции над двумя операндами, поданными на его входы A и B. На рис. 7.1 и 7.2 одним из операндов является выходное значение мультиплексора MUX, а второе считывается непосредственно с шины. Сгенерированный АЛУ результат временно запоминается в регистре Z. Последовательность операций по прибавлению содержимого регистра R1 к содержимому регистра R2 и записи результата в регистр R3 приведена ниже.

1. $R1_{out}$, Y_{in}.
2. $R2_{out}$, SelectY, Add, Z_{in}.
3. Z_{out}, $R3_{in}$.

Сигналы очередного шага активизируются на время соответствующего этому шагу такта. Все остальные сигналы в это время не активны. Так, на шаге 1 активны выход регистра R1 и вход регистра Y, поэтому содержимое регистра R1 по шине пересылается в регистр Y. На шаге 2 сигнал на управляющей линии мультиплексора устанавливается в SelectY, поэтому мультиплексор направляет содержимое регистра Y на вход A арифметико-логического устройства. В это же время содержимое регистра R2 передается на шину и через нее на вход B. Выполняемая АЛУ функция задается сигналами на его управляющих линиях. В данном случае линия Add устанавливается в 1, и в ответ АЛУ генерирует сумму двух чисел на входах A и B. Эта сумма загружается в регистр Z, входной сигнал которого активен. На шаге 3 содержимое регистра Z пересылается в результирующий регистр R3. Последняя операция пересылки не может быть выполнена на шаге 2, так как на одном тактовом цикле с шиной может быть соединен выход только одного регистра.

В этой вводной части мы предполагаем, что каждой выполняемой процессором функции соответствует специальный сигнал. В частности, отдельным управляющим сигналом задается каждая операция АЛУ (сложения, вычитания, Исключающее ИЛИ и т. д.). На практике же операции кодируются посредством меньшего количества сигналов. Например, если АЛУ может выполнять 8 операций, для их выбора достаточно трех управляющих линий. Ограничения, преимущества и недостатки принципа кодировки управляющих сигналов мы рассмотрим в разделе 7.5.1.

7.1.3. Выборка слова из памяти

Чтобы выбрать из памяти слово информации, процессор должен задать адрес этого слова и запросить операцию считывания. Причем не имеет значения, выбирается

из памяти команда программы или слово данных. Процессор помещает адрес в регистр MAR, выход которого соединен с адресными линиями шины памяти. В то же время он с помощью управляющих линий шины памяти указывает, что хочет выполнить операцию считывания. Полученные из памяти данные сохраняются в регистре MDR, откуда они могут быть пересланы в другие регистры процессора.

Соединения регистра MDR показаны на рис. 7.4. У этого регистра четыре сигнала: MDR_{in} и MDR_{out} управляют соединением с внутренней шиной, а MDR_{inE} и MDR_{outE} — соединением с внешней шиной. В представленную на рис. 7.3 схему легко добавить дополнительные соединения. В этом случае можно применить трехвходовый мультиплексор, а линия данных шины памяти может быть соединена с третьим входом. Этот вход выбирается, когда $MDR_{inE} = 1$. Второй вентиль с тремя состояниями, управляемый сигналом MDR_{outE}, может использоваться для соединения выхода триггера с шиной памяти.

В ходе операций считывания и сохранения процесс тактирования внутренней работы процессора должен координироваться с сигналами устройства, адресуемого через шину памяти. Одну внутреннюю пересылку данных процессор всегда выполняет за один такт, несмотря на то что скорость адресуемых устройств может быть очень разной. Возможно, вы помните, что современные процессоры содержат кэш-память, которая располагается на той же микросхеме, что и процессор (об этом речь шла в главе 5). Как правило, кэш отвечает на запрос чтения данных из памяти за один такт. Однако если нужные данные в кэше отсутствуют, запрос перенаправляется в основную память, и тогда происходит задержка в несколько тактов. Запрос чтения или записи может быть адресован регистру устройства ввода-вывода, адресное пространство которого отображается в основную память. Содержимое таких регистров не кэшируется, поэтому на обращение к ним всегда уходит несколько тактов.

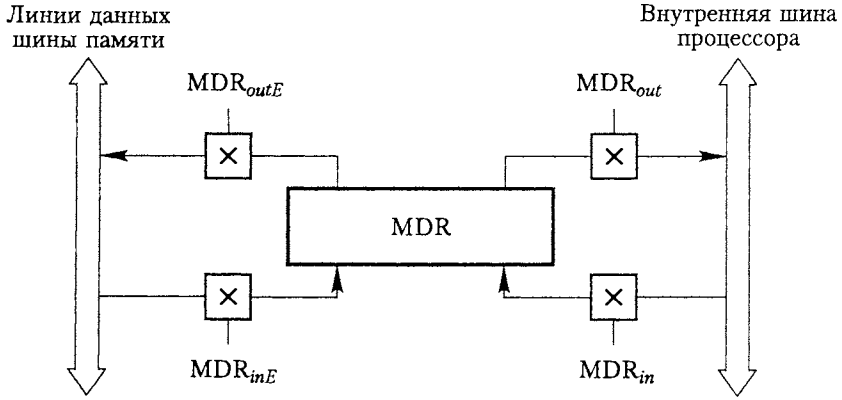


Рис. 7.4. Соединения и управляющие сигналы регистра MDR

Прежде чем вступать во взаимодействие с какими бы то ни было устройствами, процессор ждет сообщения о завершении запрошенной операции считывания.

Мы будем считать, что для этой цели используется управляющий сигнал MFC (Memory Function Complete). Адресуемое устройство устанавливает значение этого сигнала в 1, указывая тем самым, что содержимое заданного адреса прочитано и помещено на линии данных шины памяти. Вы уже встречались с такими сигналами, в частности с сигналами Slave-ready и TRDY#, при обсуждении различных типов шин в главе 4 (см. рис. 4.25 и 4.41 соответственно).

Ниже показано, как можно выполнить операцию чтения данных из памяти с помощью команды Move (R1),R2. Ее реализацию можно разбить на пять этапов.

1. Запись MAR ← [R1].
2. Начало операции чтения с шины памяти.
3. Ожидание сигнала MFC на шине памяти.
4. Загрузка в MDR данных на шине памяти.
5. Запись R2 ← [MDR].

Эти действия выполняются последовательно, но некоторые из них допускают объединение в один шаг. Каждое из указанных действий может быть завершено за один такт. Исключение составляет лишь действие 3, время выполнения которого зависит от скорости адресуемого устройства.

Для простоты изложения материала мы предполагаем, что выход регистра MAR доступен всегда. Следовательно, его содержимое всегда доступно на адресных линиях шины. Это случай, когда процессор является хозяином шины. Когда в MAR загружается новый адрес, он появляется на шине памяти в начале следующего такта, как показано на рис. 7.5. Управляющий сигнал считывания активизируется одновременно с загрузкой регистра. В ответ на этот сигнал схема интерфейса шины помещает на шину команду чтения MR. При такой организации действия 1 и 2 можно объединить в один управляющий шаг. Действия 3 и 4 тоже можно объединить, если активизировать управляющий сигнал MDR_{inE} во время ожидания ответа основной памяти. Полученные из памяти данные будут загружены в MDR в конце того такта, на котором получен сигнал MFC. На следующем такте для пересылки данных в регистр R2 активизируется линия MDR_{out}. Это означает, что операция чтения данных из памяти может быть выполнена за три шага.

1. $R1_{out}$, MAR_{in}, Read.
2. MDR_{inE}, WMFC.
3. MDR_{out}, $R2_{in}$.

Здесь WMFC — это управляющий сигнал, сообщающий управляющим схемам процессора, что они должны ждать поступления сигнала MFC.

На рис. 7.5 показано, что сигнал MDR_{inE} устанавливается в 1 на время действия команды чтения MR. В дальнейшем мы не будем явно указывать значение MDR_{inE}, поскольку оно всегда совпадает с длительностью выполнения команды MR.

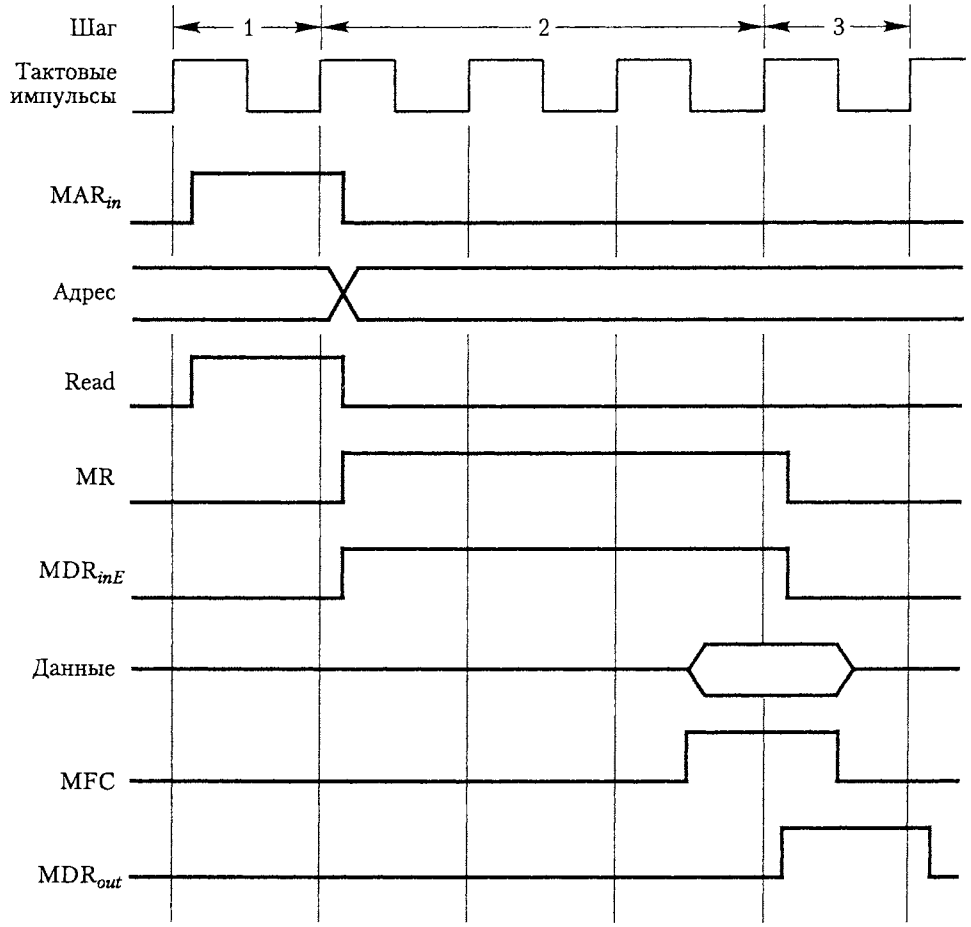


Рис. 7.5. Временная диаграмма операции чтения из памяти

7.1.4. Сохранение слова в памяти

Запись слова по заданному адресу памяти производится похожим образом. Адрес загружается в регистр MAR. Затем данные, подлежащие записи в память, загружаются в MDR и выдается команда записи. Команда Move R2,(R1) выполняется так:

1. R1$_{out}$, MAR$_{in}$.
2. R2$_{out}$, MDR$_{in}$, Write.
3. MDR$_{outE}$, WMFC.

Как и в случае операции чтения, управляющий сигнал записи указывает интерфейсной схеме шины памяти на необходимость поместить на шину команду Write. Процессор задерживается на шаге 3 до тех пор, пока не будет получен ответ MFC, означающий, что операция с памятью завершена.

7.2. Выполнение всей команды

Мы рассмотрели элементарные операции, необходимые для выполнения команды, и теперь можно свести их в единую последовательность. Рассмотрим такую команду:

Add (R3),R1

Данная команда прибавляет содержимое памяти по адресу, заданному в регистре R3, к содержимому регистра R1. Процессор это делает в четыре этапа.

1. Выборка команды.
2. Выборка первого операнда (содержимого памяти по адресу R3).
3. Выполнение сложения.
4. Загрузка результата в регистр R1.

На рис. 7.6 приведена последовательность управляющих шагов, которые необходимы для реализации этих операций процессором с единой шиной (рис. 7.1). Команда выполняется следующим образом. На шаге 1 инициируется операция выборки команды, для чего в регистр MAR загружается содержимое регистра PC, а в память направляется запрос на считывание. На управляющий вход мультиплексора MUX подается сигнал Select4, чтобы мультиплексор выбрал константу 4. Это значение прибавляется к операнду на входе B, которым является содержимое регистра PC, и результат записывается в регистр Z. На шаге 2, пока процессор ожидает ответного сигнала памяти, обновленное значение перемещается из регистра Z обратно в регистр PC. На шаге 3 выбранное из памяти слово загружается в регистр IR.

Шаг	Действие
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Рис. 7.6. Управляющая последовательность для выполнения команды Add (R3),R1

Шаги с 1 по 3 составляют фазу выборки команды, одинаковую для всех команд. Содержимое регистра IR интерпретируется дешифратором команды в начале шага 4. Это позволяет управляющей схеме активизировать управляющие сигналы для шагов с 4 по 7, составляющих фазу выполнения. На шаге 4 содержимое регистра R3 пересылается в регистр MAR и инициируется операция чтения из памяти. После этого, на шаге 5, содержимое регистра R1 для подготовки к операции сложения пересылается в регистр Y. По завершении операции чтения полученный

операнд оказывается в регистре MDR и на шаге 6 выполняется сложение. Содержимое регистра MDR передается на внутреннюю шину, а с нее — на второй вход АЛУ, для чего на мультиплексор подается сигнал SelectY. Сумма сохраняется в регистре Z, а на шаге 7 пересылается в регистр R1. И наконец, сигнал End, означающий, что выполнение команды завершено, инициирует новый цикл выборки команды и возврат к шагу 1.

Мы ни слова не сказали об управляющем сигнале Y_{in}, указанном на рис. 7.6 в перечне операций шага 2. При выполнении команды сложения нет необходимости копировать обновленное содержимое регистра PC в регистр Y. Однако в командах перехода для вычисления целевого адреса перехода необходимо использовать обновленное содержимое регистра PC. Чтобы ускорить выполнение команды Branch, это значение на шаге 2 копируется в регистр Y. А поскольку шаг 2 входит в фазу выборки, указанное действие выполняется для всех команд. Вреда от этого никакого не будет, поскольку регистр Y ни для каких других целей в данное время не используется.

7.2.1. Команды перехода

Команда перехода заменяет содержимое регистра PC целевым адресом перехода. Этот адрес обычно получают путем добавления смещения X, заданного в команде перехода, к обновленному значению регистра PC. Управляющая последовательность, реализующая команду безусловного перехода, приведена на рис. 7.7. Выполнение данной команды, как обычно, начинается с фазы выборки. Эта фаза завершается на шаге 3 загрузкой команды в регистр IR. Значение смещения извлекается из регистра IR схемой дешифрации команды, которая заодно, если нужно, выполняет расширение знака. Поскольку обновленное значение регистра PC к этому времени уже скопировано в регистр Y, смещение X передается на шину на шаге 4, после чего осуществляется операция сложения. Результат этой операции, представляющий собой целевой адрес перехода, загружается в регистр PC на шаге 5.

Шаг	Действие
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, $R1_{in}$, End

Рис. 7.7. Управляющая последовательность для команды безусловного перехода

Смещение X, задаваемое в команде перехода, обычно равно разности между целевым адресом перехода и адресом, непосредственно следующим за командой перехода. Например, если команда перехода расположена в памяти по адресу 2000 и переход следует выполнить по адресу 2050, смещение X должно быть равным 46. Чтобы понять, почему это так, нужно проанализировать управляющую

последовательность, приведенную на рис. 7.7. Приращение значения PC происходит во время фазы выборки команды, еще до того, как станет известен тип выполняемой команды. Поэтому, когда на шаге 4 вычисляется адрес перехода, значение PC уже обновлено и указывает на следующую команду в памяти.

Теперь рассмотрим условный переход. Перед загрузкой нового значения в регистр PC нужно проверить состояние кодов условий. Например, для команды Branch on negative (Branch<0) шаг 4 на рис. 7.7 необходимо заменить следующим:

$$\text{Offset-field-of-IR}_{out}, \text{ Add, } Z_{in}, \text{ if N = 0 then End}$$

Это означает, что если N = 0, то после шага 4 процессор немедленно возвращается к шагу 1. При N = 1 выполняется шаг 5, на котором в регистр PC загружается новое значение, то есть выполняется операция перехода.

7.3. Многошинная архитектура

Для иллюстрации базовых принципов организации и функционирования процессора мы использовали простую одношинную архитектуру, показанную на рис. 7.1. При такой архитектуре управляющие последовательности для выполнения команд, приведенные на рис. 7.6 и 7.7, получились довольно длинными, поскольку за один тактовый цикл по шине может пересылаться только один элемент данных. Чтобы сократить количество шагов, необходимых для выполнения команды, в большинстве современных процессоров для параллельной пересылки информации задействуется несколько внутренних каналов.

На рис. 7.8 показана трехшинная схема соединения регистров и АЛУ процессора. Все регистры общего назначения объединены в единый блок, названный *регистровым файлом*. В технологии СБИС самый эффективный способ реализации большого массива регистров заключается в объединении их в матрицу запоминающих ячеек, подобную матрице памяти с произвольным доступом (RAM), описанной в главе 5. Регистровый файл на рис. 7.8 имеет три порта. Первые два — это выходы, позволяющие одновременно обращаться к двум разным регистрам и помещать их содержимое на шины A и B. Третий порт дает возможность на том же тактовом цикле загрузить данные с шины C в третий регистр.

Шины A и B используются для пересылки исходных операндов на входы A и B АЛУ, где над ними производятся арифметические и логические операции. Результат пересылается в регистр назначения по шине C. Если нужно, АЛУ может передать один из двух входных операндов на шину C без изменения. Управляющие сигналы АЛУ, используемые для таких операций, мы обозначаем как R=A и R=B. Регистры Y и Z, показанные на рис. 7.1, при трехшинной архитектуре не нужны.

Второй особенностью архитектуры процессора, представленной на рис. 7.8, является наличие инкрементора, используемого для увеличения содержимого регистра PC на 4. Инкрементор освобождает АЛУ от регулярного выполнения этой операции, входившей в состав управляющих последовательностей, показанных на рис. 7.6 и 7.7. А вот источник константы 4 для мультиплексора по-прежнему

нужно указывать. Он может использоваться для приращения других адресов, например адресов памяти в командах групповой загрузки и сохранения.

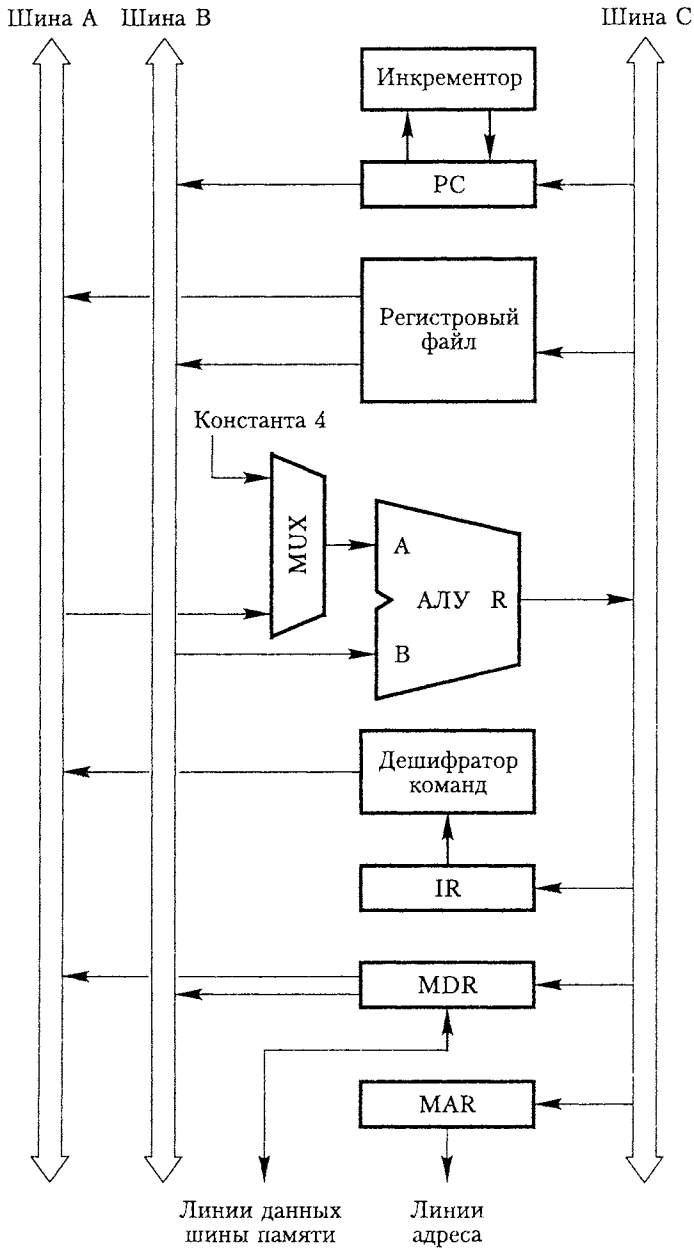


Рис. 7.8. Организация тракта данных с применением трех шин

Рассмотрим команду с тремя операндами:

Add R4,R5,R6

Управляющая последовательность действий по выполнению этой команды приведена на рис. 7.9. На шаге 1 содержимое регистра PC проходит через АЛУ в соответствии с управляющим сигналом R=B и загружается в регистр MAR для выполнения операции чтения из памяти. Одновременно с этим содержимое регистра PC увеличивается на 1. Обратите внимание на то, что в MAR загружается его исходное содержимое. Значение регистра PC увеличивается в конце такта и не влияет на содержимое регистра MAR. На шаге 2 процессор ждет сигнала MFC и загружает полученные данные в регистр MDR, после чего на шаге 3 пересылает их в регистр IR. Фаза выполнения команды состоит из единственного шага — шага 4.

Шаг	Действие
1	PC_{out}, R=B, MAR_{in}, Read, InPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Рис. 7.9. Управляющая последовательность выполнения команды Add R4, R5, R6 для трехшинной архитектуры

Благодаря увеличению числа каналов для пересылки данных количество тактов процессора, необходимых для выполнения команды, значительно сокращается.

7.4. Аппаратное управление

Для выполнения команд процессор должен генерировать соответствующие последовательности управляющих сигналов. Разработчики компьютеров справляются с этой задачей по-разному. Все возможные решения подразделяются на две основные категории: с использованием аппаратного и микропрограммного управления. Мы поговорим об обоих подходах.

Рассмотрим последовательность управляющих сигналов, приведенную на рис. 7.6. Каждый шаг этой последовательности выполняется за один такт. Для отслеживания управляющих шагов можно применить специальный счетчик, как показано на рис. 7.10. Каждое значение этого счетчика соответствует одному управляющему шагу. При выборе управляющих сигналов учитываются следующие данные:

- содержимое счетчика управляющих шагов;
- содержимое регистра команды;
- значения флагов кодов условий;
- внешние входные сигналы, такие как сигналы запросов прерывания MFC.

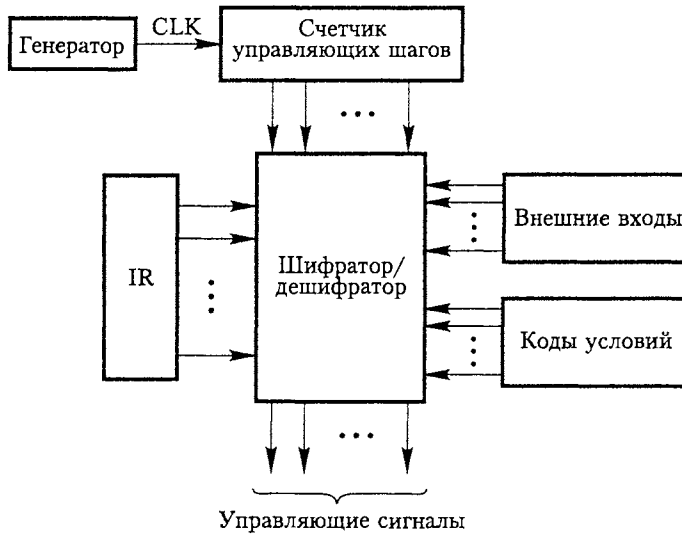


Рис. 7.10. Организация управляющего блока

Знакомство с внутренней структурой управляющего блока мы начнем с рассмотрения его упрощенной схемы. Шифратор/дешифратор (рис. 7.10) представляет собой комбинационную схему, генерирующую управляющие сигналы на основе состояния всех ее входов. Разделив функции шифрации и дешифрации, мы получим более детальную блок-схему (рис. 7.11). Каждому шагу или временному интервалу управляющей последовательности соответствует отдельная сигнальная линия дешифратора шага. Аналогичным образом, на выходе дешифратора команд имеется отдельная линия для каждой машинной команды. Для любой команды, загруженной в регистр IR, одна из выходных линий от INS_1 до INS_m устанавливается в 1, а все остальные линии — в 0. (Конструкция дешифраторов подробно описана в приложении А.) Входные сигналы шифратора, как показано на рис. 7.11, объединяются для формирования отдельных управляющих сигналов Z_{in}, PC_{out}, Y_{in}, Add, End и т. д. Пример формирования управляющего сигнала Z_{in} для архитектуры процессора, показанной на рис. 7.1, приведен на рис. 7.12. Эта схема реализует логическую функцию

$$Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \ldots \qquad (7.1)$$

В течение временного интервала T_1 сигнал Z_{in} генерируется для всех команд, в течение интервала T_6 — только для команды сложения, в течение интервала T_4 — для команды безусловного перехода и т. д. Логическая функция для Z_{in} построена на основе логических последовательностей, приведенных на рис. 7.6 и 7.7. Еще один пример формирования управляющего сигнала представлен на рис. 7.13. Это схема, генерирующая сигнал End на основе логической функции

$$End = T_7 \cdot Add + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{N}) \cdot BRN + \ldots \qquad (7.2)$$

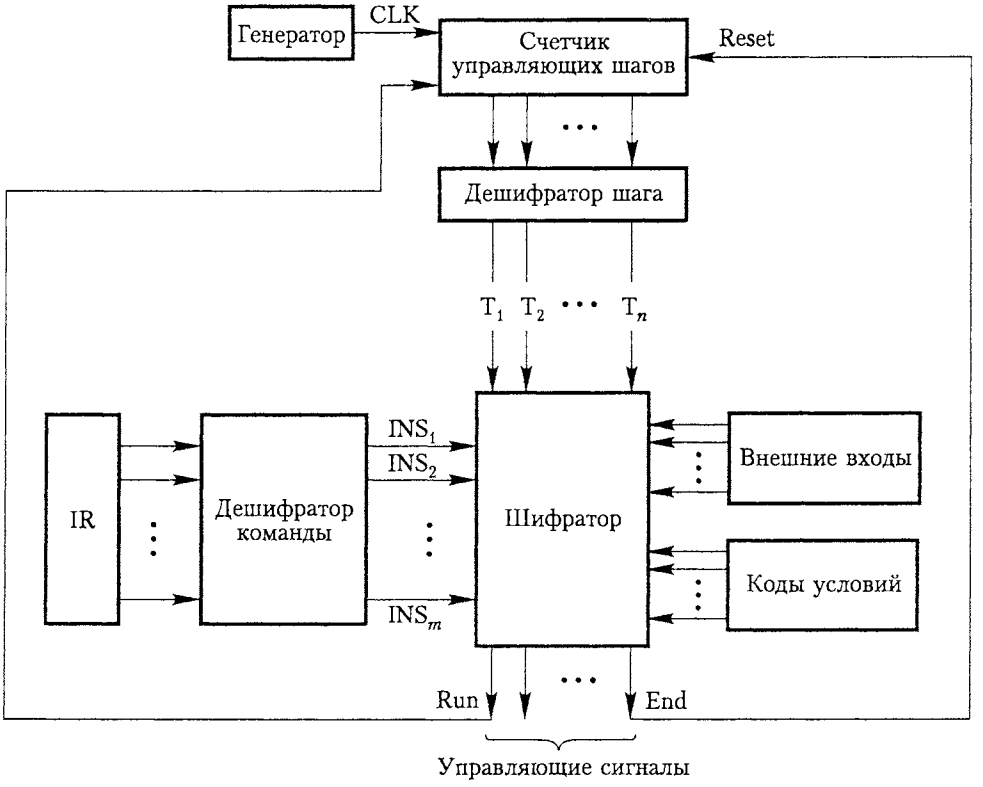


Рис. 7.11. Организация управляющего блока

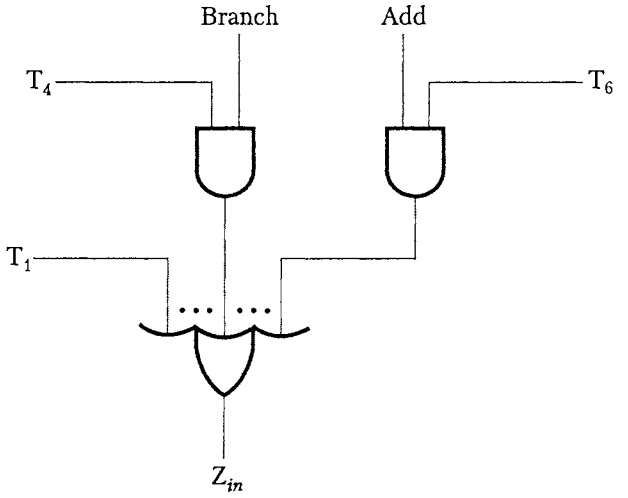


Рис. 7.12. Формирование управляющего сигнала Z_{in} для процессора, схема которого приведена на рис. 7.11

Сигнал End начинает новый цикл выборки команды, сбрасывая счетчик управляющих шагов в начальное состояние. На рис. 7.11 показан еще один управляющий сигнал, названный Run. Когда он установлен в 1, в конце каждого тактового цикла значение счетчика увеличивается на 1. Если же Run становится равным 0, отсчет шагов прекращается. Так происходит в случае выдачи сигнала WMFC, после которого процессор должен дожидаться ответного сигнала блока памяти.

Управляющие схемы, приведенные на рис. 7.10 и 7.11, можно рассматривать в качестве автомата с конечным числом состояний (конечный автомат), который на каждом такте переходит из одного состояния в другое, определяемое содержимым регистра команды, кодами условий и внешними входами. Выходами такого автомата являются управляющие сигналы. Управляемая им последовательность операций определяется связями между логическими элементами. Контроллер, в котором используется данный подход, может работать с очень высокой скоростью. Однако ему недостает гибкости, а объем и сложность системы команд, которая может быть в нем реализована, очень ограниченны.

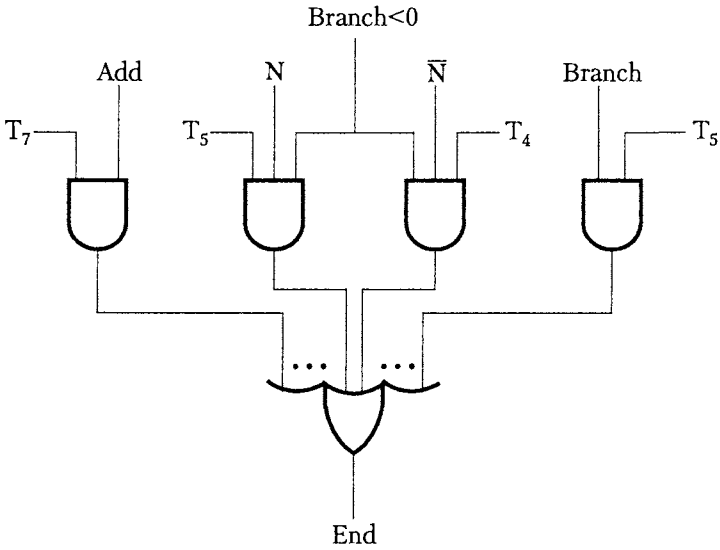


Рис. 7.13. Формирование управляющего сигнала End

7.4.1. Структура процессора

Один из вариантов обобщенной структуры процессора представлен на рис. 7.14. Здесь имеется блок, выбирающий команду из кэша команд, а если ее там нет — то из основной памяти. Два отдельных процессорных блока предназначены для обработки целочисленных данных и данных с плавающей запятой. Каждый из этих блоков может быть организован так, как показано на рис. 7.8. Кэш данных располагается между этими блоками и основной памятью. Технология разделения кэша

команд и кэша данных широко применяется во многих современных процессорах. Есть и такие процессоры, в которых единственный кэш содержит и команды и данные. Процессор соединяется с системной шиной и остальной частью компьютера посредством шинного интерфейса.

Хотя на рис. 7.14 показано по одному целочисленному блоку и блоку для чисел с плавающей запятой, в процессоре может быть несколько устройств каждого из этих типов, обеспечивающих параллельное выполнение большего количества вычислений. Способы наиболее эффективной организации устройств, способные ускорить выполнение команд, описаны в главе 8.

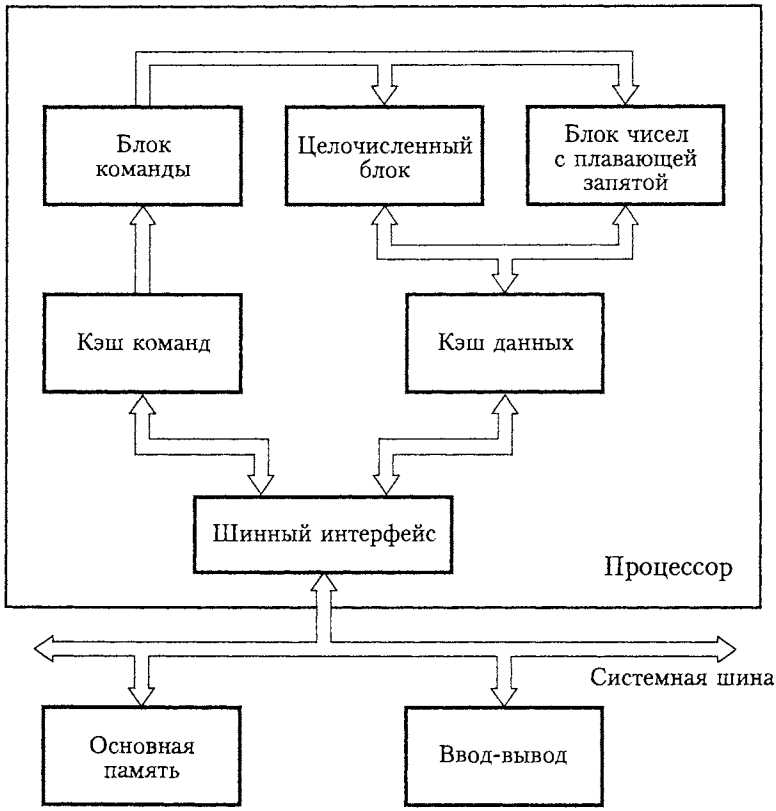


Рис. 7.14. Блок-схема всего процессора

7.5. Микропрограммное управление

В разделе 7.4 было показано, как с помощью счетчика управляющих шагов и схемы шифратора/дешифратора генерируются внутренние управляющие сигналы процессора. Теперь мы поговорим об альтернативной схеме, называемой *микропрограммным управлением*, согласно которой сигналы генерируются программой, подобной написанным на машинном языке.

Прежде всего мы введем несколько стандартных терминов. *Управляющее слово* (Control Word, CW) — это слово, отдельные биты которого представляют различные управляющие сигналы, генерируемые схемой, которая приведена на рис. 7.11

Каждый шаг управляющей последовательности команды определяет уникальную комбинацию нулей и единиц в управляющем слове. Для примера на рис. 7.15 приведены управляющие слова, соответствующие семи шагам последовательности, показанной на рис. 7.6. Мы предполагаем, что сигнал SelectY представлен значением 0 управляющего входа, а сигнал Select4 — значением 1. Последовательность управляющих слов, соответствующих управляющей последовательности конкретной машинной команды, составляет *микропрограмму* этой команды, а отдельное управляющее слово микропрограммы называется *микрокомандой.*

Микрокоманда	...	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	...
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Рис. 7.15. Пример микрокоманд, реализующих управляющую последовательность, которая представлена на рис. 7.6

Микропрограммы всех команд системы команд процессора хранятся в специальной памяти, называемой *управляющей памятью.* Прежде чем сгенерировать сигналы команды, управляющий блок последовательно считывает из управляющей памяти слова соответствующей микропрограммы. Структурная схема этого блока показана на рис. 7.16. Последовательное чтение слов из управляющей памяти обеспечивается счетчиком микропрограммы μPC. При каждой загрузке в регистр IR новой команды в μPC загружается выходное значение блока, названного на схеме генератором начального адреса. После этого на очередном такте выполняется автоматическое приращение содержимого μPC для выбора из управляющей памяти очередной команды. Благодаря этому управляющие сигналы поступают в разные части процессора в правильной последовательности.

Управляющий блок обладает одной важной функцией, которую нельзя реализовать при такой его структуре, как показано на рис. 7.16. Речь идет о ситуации, когда управляющий блок должен проанализировать состояние кодов условий или внешних входов, чтобы выбрать одно из альтернативных действий. В случае аппаратного управления для обработки такой ситуации используется соответствующая логическая функция (см. формулу 7.2), интегрированная в схему шифратора.

При микропрограммном управлении необходимые проверки выполняются с помощью микрокоманд условного перехода. Кроме целевых адресов в этих микрокомандах задаются внешние входы, управляющие коды и разряды регистра команды, от значений которых зависит выбор адреса перехода.

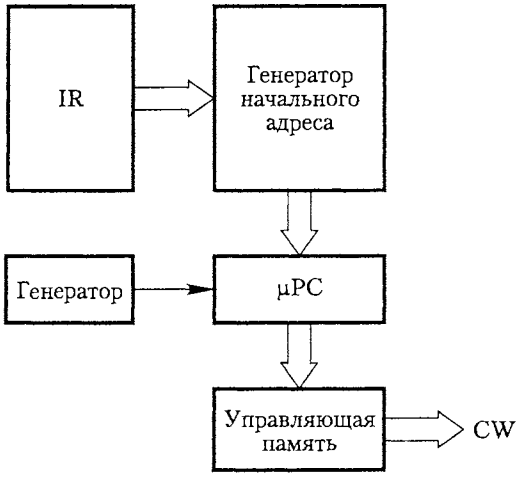


Рис. 7.16. Базовая организация управляющего микропрограммного блока

Команда Branch<0 в этом случае может быть реализована с помощью микропрограммы, приведенной на рис. 7.17. После загрузки данной команды в регистр IR микрокоманда перехода передает управление соответствующей микропрограмме, которая в нашем примере начинается по адресу управляющей памяти 25. Этот адрес выдается генератором начального адреса, показанным на рис. 7.16. Микрокоманда по адресу 25 проверяет разряд N кодов условий. Если в нем содержится 0, выполняется переход по адресу 0 для выборки новой машинной команды. В противном случае выполняется микрокоманда по адресу 26, которая помещает целевой адрес перехода в регистр Z (см. шаг 4 на рис. 7.7). Микрокоманда по адресу 27 загружает этот адрес в регистр PC.

Шаг	Действие
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Переход к начальному адресу соответствующей микропрограммы
...	...
25	If N = 0, then переход к микрокоманде 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Рис. 7.17. Микропрограмма для выполнения команды Branch<0

Для поддержки ветвления в микропрограммах управляющий блок должен быть таким, как показано на рис. 7.18. На генератор начального адреса теперь возлагается дополнительная функция — генерирование адреса перехода. По указанию микрокоманды этот блок загружает в счетчик μPC новый адрес. Для поддержки условных переходов на входы данного блока подаются сигналы с внешних входов, коды условий и содержимое регистра команд. В этом управляющем блоке после каждой выборки микрокоманды из управляющей памяти происходит приращение значения μPC. Но возможны исключения.

1. Если в регистр IR загружается новая команда, в счетчик μPC загружается начальный адрес ее микропрограммы.
2. Когда встречается микрокоманда Branch и удовлетворяется условие перехода, в μPC загружается адрес перехода.
3. В случае применения микрокоманды End в μPC загружается адрес первого управляющего слова микропрограммы фазы выборки команды (на рис. 7.17 это адрес 0).

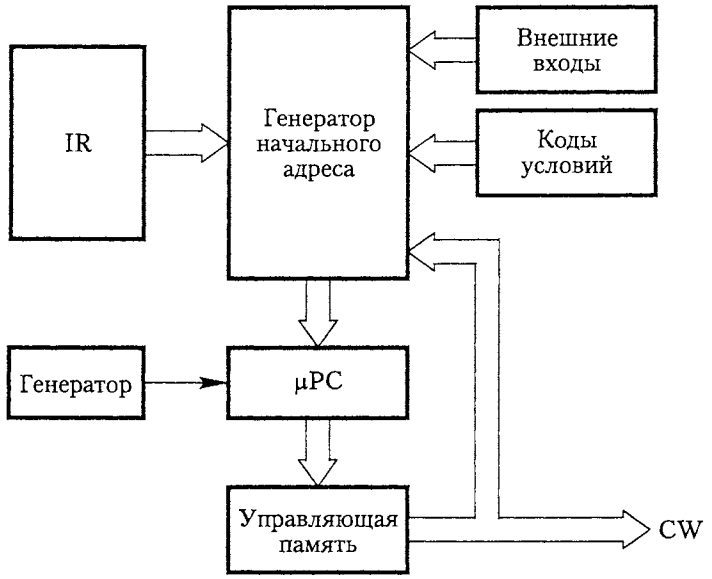


Рис. 7.18. Организация управляющего блока, поддерживающего условные переходы в микропрограммах

7.5.1. Микрокоманды

Рассмотрев схему управления процессом выполнения микропрограмм, мы можем обратиться к формату отдельных микрокоманд. Самый простой способ структурирования микрокоманд заключается в назначении каждому управляющему сигналу одного разряда (рис. 7.15).

Однако у этой схемы имеется серьезный недостаток: если каждому управляющему сигналу назначить отдельный разряд, микрокоманды получатся очень длинными, поскольку количество сигналов обычно достаточно велико. Более того, в единицу устанавливается всего несколько разрядов (соответствующих активным вентилям), так что выделенное для микрокоманд пространство используется очень нерационально. Давайте еще раз обратимся к простому процессору, показанному на рис. 7.1, и предположим, что он содержит только четыре регистра общего назначения, а именно R0, R1, R2 и R3. Некоторые соединения в этом процессоре временно активны, как, например, соединение выхода регистра IR со схемами дешифратора и обоими входами АЛУ. Для остальных соединений с различными регистрами требуется 20 управляющих сигналов. Кроме того, нужны еще и дополнительные управляющие сигналы, не показанные на этом рисунке, в том числе сигналы Read, Write, Select, WMFC и End. И еще нам следует определить функцию, которая должна быть выполнена АЛУ. Предположим, что всего таких функций 16, среди которых функции Add, Subtract, AND и XOR. Набор этих функций зависит от конкретного АЛУ и необязательно один к одному совпадает с кодами операций машинных команд. Итого, нам необходимо 42 управляющих сигнала.

При использовании описанной выше простой схемы кодирования микрокоманд для каждой из них потребуется 42 бита. Однако длину микрокоманд можно сократить. Большинство сигналов не используется одновременно, а многие из них даже являются взаимоисключающими. Например, за один раз может быть активизирована только одна функция АЛУ. Источник пересылки данных должен быть уникальным, поскольку на шину нельзя одновременно поместить содержимое двух разных регистров. Не могут быть одновременно активными и сигналы Read и Write, инициирующие чтение из памяти и запись в память. Из всего сказанного следует, что сигналы можно сгруппировать таким образом, чтобы каждую группу составляли взаимоисключающие сигналы. Тогда в любой микрокоманде будет задаваться только одна *микрооперация* из группы. Для представления сигналов группы можно использовать специальную схему двоичного кодирования. Скажем, для представления 16 функций АЛУ достаточно четырех разрядов. В одну группу, в частности, можно объединить управляющие сигналы PC_{out}, MDR_{out}, Z_{out}, $Offset_{out}$, $R0_{out}$, $R1_{out}$, $R2_{out}$, $R3_{out}$ и $TEMP_{out}$. Любой из них нетрудно будет выбрать с помощью уникального 4-разрядного кода.

Оставшиеся сигналы тоже можно объединить в группы. На рис. 7.19 дан пример формата микрокоманд, в котором каждой группе сигналов соответствует поле длины, достаточной для размещения кода сигнала в группе. Для большинства полей должен быть определен еще один код, означающий, что ни один из сигналов группы не активен. Например, все нули в поле F1 означают, что ни один из регистров, которые могут быть заданы в этом поле, не должен помещать свое содержимое на шину. Такой код нужен не для всех полей. В частности, поле F4 занимает четыре разряда, определяющих одну из 16 выполняемых АЛУ операций. Поскольку дополнительный код в нем не задается, АЛУ активно при выполнении любой микрокоманды. Его действиями система управляет через регистр Z, который загружается только при активизации сигнала Z_{in}.

F1	F2	F3	F4	F5	

F1 (4 бита)	F2 (3 бита)	F3 (3 бита)	F4 (4 бита)	F5 (2 бита)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$		
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	1111: XOR	
0101: $R1_{out}$	101: $R1_{in}$		16 функций АЛУ	
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6	F7	F8	...

F6 (1 бит)	F7 (1 бит)	F8 (1 бит)
0: SelectY	0: No action	0: Continue
1: Select4	1: WMFC	1: End

Рис. 7.19. Примерный формат микрокоманд с кодировкой полей (No transfer — пересылка не осуществляется; No action — никакое действие не производится; Continue — микропрограмма продолжает выполняться)

Для группировки управляющих сигналов в полях необходимы дополнительные схемы, поскольку коды здесь должны преобразовываться в отдельные управляющие сигналы. Но стоимость таких дополнительных схем будет с лихвой компенсирована за счет уменьшения количества разрядов в каждой микрокоманде, а значит, уменьшением объема управляющей памяти. В микропрограмме, представленной на рис. 7.19, для хранения кодов 42 сигналов достаточно 20 разрядов.

До сих пор речь шла о группировке и кодировании только взаимоисключающих управляющих сигналов. Эту идею можно расширить, пронумеровав наборы управляющих сигналов во всех возможных микрокомандах. Каждой реальной комбинации активных управляющих сигналов может быть присвоен отдельный код, представляющий микрокоманду. Такое полное кодирование позволит еще больше сократить длину микрослов, но усложнит схемы дешифрации микрокоманд.

Использование сильно закодированных схем с компактными кодами, определяющими только то небольшое количество управляющих сигналов, которое действительно имеется в микрокоманде, называется *вертикальной организацией микрокоманд*. А приведенная на рис. 7.15 схема с минимальным кодированием,

в соответствии с которой каждая микрокоманда управляет большим количеством ресурсов, называется *горизонтальной организацией*. Принцип горизонтальной организации полезен в тех случаях, когда главной задачей конструкторов является достижение максимальной скорости работы процессора и при этом архитектура компьютера допускает параллельное использование ресурсов. Вертикальная организация замедляет работу, поскольку для выполнения управляющих функций требуется большее количество микрокоманд. И хотя микрокоманды в этом случае имеют меньшую длину, это вовсе не значит, что общее количество битов управляющей памяти также будет меньшим. Преимущество вертикального подхода заключается не в уменьшении объема микропрограмм, а в снижении аппаратных затрат на обработку микрокоманд.

Горизонтальная и вертикальная организация представляют два принципиально различных способа организации микропрограммного управления. Существует и множество промежуточных схем с разной степенью кодирования. Представленная на рис. 7.19 схема имеет горизонтальную организацию, поскольку в ней сгруппированы только взаимоисключающие микрооперации. Она никак не зависит от способности процессора выполнять микрокоманды параллельно.

Хотя мы рассмотрели лишь небольшое подмножество управляющих сигналов, оно в достаточной мере отражает работу процессора. Были опущены некоторые детали, не существенные для понимания принципов функционирования процессора.

7.5.2. Управление выполнением микропрограмм

Для реализации микропрограммы, приведенной на рис. 7.15, достаточно выполнять все микрокоманды последовательно, за исключением одного случая, когда требуется осуществить переход в конце фазы выборки команды. И если машинные команды реализованы в виде подобных микропрограмм, для их выполнения вполне подходит показанная на рис. 7.18 управляющая структура, в которой последовательность микрокоманд определяется значением счетчика μPC. Выполнение микропрограммы начинается с дешифрации машинной команды и загрузки начального адреса соответствующей микропрограммы в μPC. Ветвление в микропрограммах реализуется с помощью специальных микрокоманд, определяющих адрес перехода — подобно тому, как это делается в программах машинного уровня.

При таком подходе написание микропрограмм — довольно простая задача, для реализации которой можно применять стандартные технологии разработки программного обеспечения. Однако имеется два серьезных недостатка. Во-первых, когда для каждой машинной команды используется отдельная микропрограмма, их общее количество получается очень большим, а следовательно, для данного подхода требуется очень большая управляющая память. А с учетом того, что в большинстве машинных команд могут применяться разные режимы адресации, команд с различными комбинациями режимов адресации получается еще больше. Написание отдельной микропрограммы для каждой такой комбинации часто сопровождается дублированием их общих фрагментов. Поэтому желательно организовать микропрограммы таким образом, чтобы общие фрагменты использовались ими совместно. Такие микропрограммы должны содержать больше команд

перехода, передающих управление между их различными частями. Здесь становится очевидным второй недостаток описанного подхода, заключающийся в увеличении времени выполнения микропрограммы за счет частых переходов.

Рассмотрим более сложный пример функционально полной машинной команды. В главе 2 обсуждались команды типа

Add src,Rdst

складывающие исходный операнд с содержимым регистра Rdst и помещающие сумму обратно в Rdst. Предположим, что исходный операнд можно задавать с помощью четырех режимов адресации, а именно регистрового, автоинкрементного, автодекрементного и индексного, а также посредством их косвенных форм. На примере данной команды, используя процессор, структура которого показана на рис. 7.1, мы продемонстрируем возможный способ реализации микропрограммы.

Чтобы облегчить восприятие излагаемого материала, мы привели на рис. 7.20 блок-схему микропрограммы. Каждый прямоугольник в ней соответствует микрокоманде, управляющей пересылкой данных или операцией, указанной в этом прямоугольнике. Микрокоманды расположены по адресам, заданным в восьмеричном формате над правым верхним углом каждого прямоугольника. Любая восьмеричная цифра представляет три бита. Мы воспользовались этой системой счисления для указания адресов только потому, что она позволяет более компактно представить двоичные числа. Большая часть схемы понятна сама по себе, но некоторые детали требуют пояснения. Однако прежде чем перейти к их детальному описанию, нам нужно рассмотреть еще несколько важных вопросов.

Модификация адреса перехода с использованием технологии логического сложения разрядов

Судя по представленной на рис. 7.20 микропрограмме, переход не всегда выполняется по единственному целевому адресу. Это прямое следствие объединения нескольких микропрограмм для совместного использования их общих частей. Обратите внимание на точку, обозначенную на рисунке буквой α. В этой точке нужно выбрать одно из действий, необходимых для прямой или косвенной адресации. Если операнд команды задан посредством косвенной адресации, его выборку из памяти выполняет микрокоманда, находящаяся по адресу 170. Если же используется прямая адресация, эту команду можно обойти и сразу перейти к команде, расположенной по адресу 171. Самый эффективный способ обхода микрокоманды по адресу 170 заключается в использовании микрокоманды перехода по адресу 170 и вентиля ИЛИ, который в случае прямой адресации изменяет младший разряд данного адреса на 1. Эта технология модификации адреса перехода называется *логическим сложением разрядов*.

Альтернативное решение состоит в использовании двух микрокоманд условного перехода по адресам 123, 143 и 166. Кроме того, в микрокоманду условного перехода можно включить два адресных поля, одно для косвенной адресации, а другое — для прямой. Но обе эти альтернативы менее эффективны, чем логическое сложение разрядов.

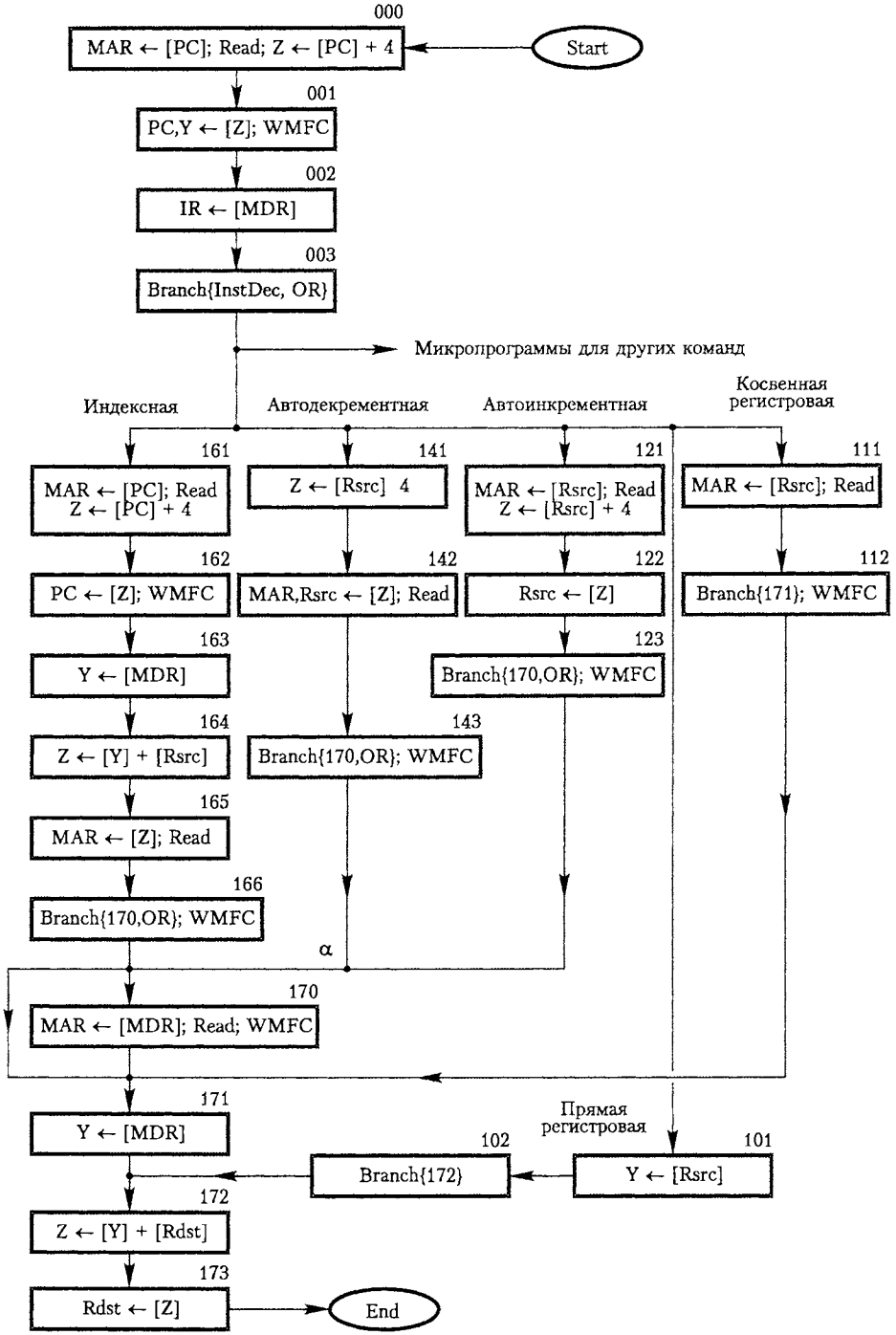


Рис. 7.20. Схема микропрограммы для выполнения команды Add src,Rdst

7.5.3. Адресация с сильным ветвлением

После команды, расположенной на рис. 7.20 по адресу 003, микропрограмма сильно разветвляется. Дешифратор команды, обозначенный на рисунке как InstDec, генерирует начальный адрес микропрограммы, которая реализует команду, только что загруженную в регистр IR. В нашем примере в указанном регистре содержится команда Add, для которой дешифратор команды генерирует адрес микрокоманды, расположенной по адресу 101. Однако этот адрес не загружается в счетчик микропрограммы без изменения.

Исходный операнд команды Add может быть задан в одном из нескольких адресных режимов, которым на рисунке соответствуют пять направлений перехода. Речь идет об индексной, автодекрементной, автоинкрементной, прямой регистровой и косвенной регистровой адресации. Описанную выше технологию логического сложения разрядов в этой точке можно применить для замены адреса 101 одним из пяти возможных значений: 161, 141, 121, 101 или 111, в зависимости от указанного в команде режима адресации.

Использование WMFC

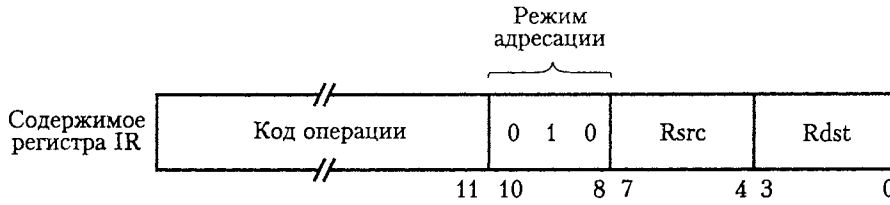
Мы предполагаем, что в микрокоманде перехода можно задать команду ожидания сигнала MFC — команду WMFC (Wait for MFC). Она используется, в частности, в микрокоманде по адресу 112, вызывающей переход к микрокоманде по адресу 171. С объединением этих двух операций связана одна проблема. Появление сигнала WMFC означает, что для завершения микрокоманды может потребоваться несколько таковых циклов. Если переход произойдет на первом такте, микрокоманда по адресу 171 будет выбрана и выполнена раньше времени. Для того чтобы этого не случилось, переход должен быть выполнен лишь после окончания операции пересылки данных из памяти, то есть сигнал WMFC должен запрещать любые изменения значений счетчика микропрограммы на все время ожидания.

Детальный анализ микропрограммы

Вернемся к представленной на рис. 7.20 программе еще раз и поговорим о ее работе более детально. Начнем с ситуации, когда исходный операнд задан в автоинкрементном режиме, то есть когда выполняется команда

Add (Rsrc)+,Rdst

где Rsrc и Rdst — регистры общего назначения. На рис. 7.21 приведен полный текст микропрограммы для выборки и выполнения этой команды. Мы предполагаем, что в команде имеется 3-разрядное поле, в котором задается режим адресации исходного операнда. Значения 11, 10, 01 и 00 в разрядах 10 и 9 соответствуют индексной, автодекрементной, автоинкрементной и регистровой адресации. Для каждого из этих режимов разряд 8 определяет косвенную версию. Например, значение 010 в поле режима соответствует прямой автоинкрементной адресации, а значение 011 — косвенной. Кроме того, мы делаем предположение, что в процессоре имеется 16 регистров, которые могут использоваться для адресации, и что каждый из этих регистров задается с помощью 4-разрядного кода. Таким образом, для определения исходного операнда достаточно указать значение в поле режима и номер регистра, заданного в разрядах от 0 до 3.



Адрес (восьмеричный)	Микрокоманда
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {μPC ← 101 (из дешифратора команды); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow \overline{[IR_{10}]} \cdot \overline{[IR_9]} \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$,
123	μBranch {μPC ← 170; $\mu PC_0 \leftarrow \overline{[IR_8]}$, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Рис. 7.21. Микрокоманда для выполнения команды Add(Rsrc)+,Rdst (Микрокоманда по адресу 170 для этого адресного режима не выполняется)

Поскольку для определения местоположения исходных и результирующих операндов можно задействовать любой из 16 регистров общего назначения, ссылки на соответствующие управляющие сигналы в микрокомандах обозначаются как $Rsrc_{out}$, $Rsrc_{in}$, $Rdst_{out}$ и $Rdst_{in}$. Дешифратор, соединенный с адресными полями Rsrc и Rdst регистра IR, преобразует эти сигналы в сигналы пересылки для конкретных регистров. Это означает, что дешифрация микрокоманды выполняется в два этапа. После первой дешифрации поля Rsrc или Rdst регистра IR становится известно, что в операции участвует один из регистров общего назначения. После этого с помощью значения на выходе дешифратора содержимое соответствующего поля перенаправляется во второй дешифратор, генерирующий сигналы для выбора одного из регистров общего назначения R0–R15.

В рассматриваемой микропрограмме (рис. 7.20) объединены микропрограммы для всех возможных режимов адресации, в результате чего получилась структура с большим количеством точек ветвления. В примере, приведенном на рис. 7.21, имеются две точки ветвления, для которых нужны две микрокоманды перехода. Для каждого режима адресации выражение в фигурных скобках указывает на адрес

перехода, который должен быть загружен в регистр μPC, и на то, что этот адрес нужно модифицировать с использованием схемы логического сложения разрядов. Для примера возьмем микрокоманду, расположенную по адресу 123. В ее исходной версии задан переход к микрокоманде по адресу 170, которая производит еще одну выборку из основной памяти, соответствующую косвенной адресации. В случае прямой адресации микропрограмма обходит команду выборки, для чего выполняется логическая операция OR над инвертированным разрядом косвенной адресации в поле адреса исходного операнда (разряд 8 в регистре IR) и разрядом 0 регистра μPC.

Еще одним примером использования технологии логического сложения разрядов может служить микрокоманда, находящаяся по адресу 0003. В зависимости от режима адресации исходного операнда должен быть выбран один из пяти начальных адресов микропрограммы, реализующей команду Add. Эти адреса различаются только средней восьмеричной цифрой. Поэтому для осуществления нужного перехода можно путем логического сложения разрядов модифицировать среднюю восьмеричную цифру адреса микропрограммы 101, полученного от дешифратора команды. Три разряда, которые посредством операции OR нужно сложить с этой цифрой, предоставляются схемой дешифратора, соединенной с полем режима адресации исходного операнда (разряды 8, 9 и 10 регистра IR). Адреса микрокоманд выбираются таким образом, чтобы было легче выполнить эту модификацию; разряды 4 и 5 регистра μPC устанавливаются непосредственно из разрядов 9 и 10 регистра IR. Это позволяет выбрать нужную микрокоманду для любого из адресных режимов, кроме одного. Косвенный регистровый режим обрабатывается путем установки разряда 3 регистра μPC в 1 при условии, что логическое произведение $[\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$ равно 1. Косвенная регистровая адресация обрабатывается особым образом, потому что это единственный косвенный режим, в котором не используется микрокоманда по адресу 170.

7.5.4. Микрокоманды с полем следующего адреса

Микропрограмма, представленная на рис. 7.20, содержит несколько микрокоманд перехода, которые не выполняют никакой полезной работы — они нужны только для определения адреса следующей микрокоманды. Поэтому их наличие замедляет работу компьютера. Ситуация становится еще хуже, если дело касается других микропрограмм. Количество микрокоманд перехода отчасти увеличивается из-за невозможности присвоить последовательные адреса даже тем микрокомандам, которые обычно выполняются в одном и том же порядке.

Эта проблема заставляет нас пересмотреть технологию управления последовательностью выполнения микрокоманд, в основе которой лежит применение счетчика μPC. Эффективной альтернативой использованию счетчика является включение в каждую микрокоманду поля адреса, указывающего местоположение следующей выбираемой микрокоманды. Это означает, что каждая микрокоманда одновременно становится и командой перехода.

За гибкость такого подхода приходится расплачиваться созданием дополнительных разрядов поля адреса. Чтобы понять, что это означает для реального компьютера, нужно проанализировать следующий факт. В типичном компьютере полный набор микропрограмм не может содержать более 4 × 1024 микрокоманд, каждая из которых будет занимать от 50 до 80 битов. Это значит, что длина адресного поля должна составлять 12 битов. Таким образом, одна шестая часть управляющей памяти процессора будет занята адресами. И даже если потребуется большее количество микропрограмм, адресное поле увеличится незначительно.

Наиболее очевидное преимущество описанного подхода заключается в том, что он практически ликвидирует отдельные микрокоманды перехода. Более того, остается меньше ограничений на присвоение микрокомандам адресов. Так что несмотря на некоторые недостатки, эти преимущества придают данному подходу еще большую привлекательность. Так как каждая команда содержит адрес следующей команды, отпадает необходимость в счетчике, вычисляющем последовательные адреса. Поэтому μPC заменяется *регистром адреса микрокоманды* μAR, который загружается из поля следующего адреса при выполнении очередной микрокоманды. Новая управляющая структура, поддерживающая такой режим выполнения команд и логическое сложение разрядов, показана на рис. 7.22. Биты следующего адреса через вентили OR пересылаются в mAR, благодаря чему становится возможным модифицировать адрес на основе данных в регистре IR, внешних входов и кодов условий. Схемы дешифрации генерируют начальный адрес заданной микропрограммы на основе кода операции в указанном регистре.

Далее мы поговорим о том, как можно реализовать показанную на рис. 7.21 микропрограмму с использованием управляющей структуры, которую вы видите на рис. 7.22. Нам нужно добавить несколько управляющих сигналов, которые не включены в формат микрокоманды, представленной на рис. 7.19. Вместо явных ссылок на регистры R0–R15 мы применяем имена Rsrc и Rdst, которые можно преобразовать в реальные управляющие сигналы на основе значений полей исходного и результирующего операндов в регистре IR. Для осуществления перехода посредством логического сложения разрядов нужно, чтобы в микрокоманды были включены соответствующие команды. В программе на рис. 7.20 логическое сложение разрядов нужно применить к микрокоманде, расположенной по адресу 003, что позволит определить адрес следующей микрокоманды на основе режима адресации исходного операнда. Режим адресации задается разрядами 8–10 регистра команды, как показано на рис. 7.21. Предположим, что операция логического сложения разрядов управляется сигналом OR_{mode}. В микрокомандах по адресам 123, 143 и 166 логическое сложение разрядов дает возможность определить, применяется ли косвенная адресация исходного операнда. Для этой цели мы задействуем сигнал OR_{indsrc}. Для упрощения эти сигналы идентифицируются отдельными разрядами микрокоманды. Еще один разряд микрокоманды указывает, когда выходной сигнал дешифратора команды должен быть направлен в μAR. К тому же каждая микрокоманда содержит 8-разрядное поле адреса следующей микрокоманды. Полный формат микрокоманд описанного типа приведен на рис. 7.23. Он является расширенным вариантом формата, представленного на рис. 7.19.

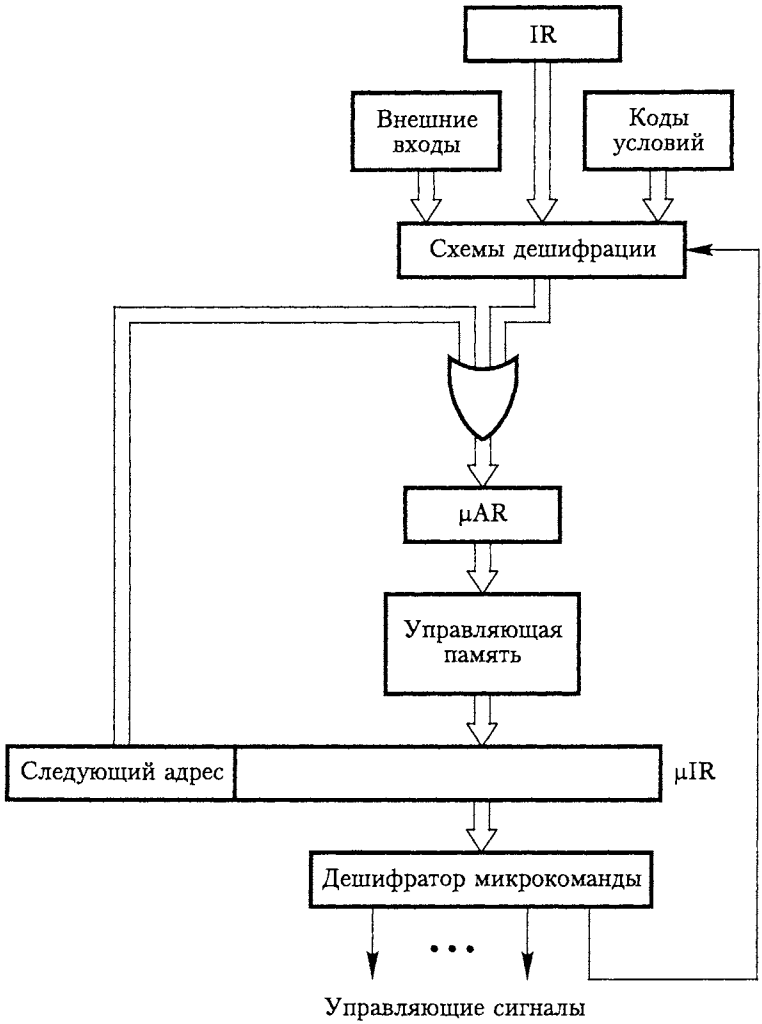


Рис. 7.22. Управление последовательностью выполнения микрокоманд

Используя подобные микрокоманды, приведенную на рис. 7.21 микропрограмму можно переписать так, как показано на рис. 7.24. В переработанной микропрограмме стало на одну микрокоманду меньше. Микрокоманда перехода по адресу 123 теперь объединена с предшествующей ей микрокомандой. Когда последовательностью выполнения микрокоманд управляет счетчик μPC, для записи в него начального адреса микрокоманды, выбирающей следующую машинную команду, применяется сигнал End. В нашем примере это начальный адрес 000_8. Однако новая микропрограмма не заканчивается сигналом End. При такой организации выполнения микропрограмм начальный адрес не задается механизмом сброса счетчика, запускаемым сигналом End, а явно указывается в поле F0.

F0	F1	F2	F3
F0 (8 битов)	F1 (3 бита)	F2 (3 бита)	F3 (3 бита)
Адрес следующей микрокоманды	000: No transfer 001: PC_{out} 010: MDR_{out} 011: Z_{out} 100: $Rsrc_{out}$ 101: $Rdst_{out}$ 110: $TEMP_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 011: Z_{in} 100: $Rsrc_{in}$ 101: $Rdst_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}

F4	F5	F6	F7
F4 (4 бита)	F5 (2 бита)	F6 (1 бит)	F7 (1 бит)
0000: Add 0001: Sub ⋮ 1111: XOR	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC

F8	F9	F10
F8 (1 бит)	F9 (1 бит)	F10 (1 бит)
0: NextAdrs 1: InstDec	0: No action 1: OR_{mode}	0: No action 1: OR_{indsrc}

Рис. 7.23. Формат микрокоманд в примере, приведенном в разделе 7.5.3 (No transfer — пересылка не выполняется; No action — никакое действие не выполняется; NextAdrs — следующий адрес; InstDec — дешифратор команды)

Восьмеричный адрес	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
000	00000001	001	011	001	0000	01	1	0	0	0	0
001	00000010	011	001	100	0000	00	0	1	0	0	0
002	00000011	010	010	000	0000	00	0	0	0	0	0
003	00000000	000	000	000	0000	00	0	0	1	1	0
121	01010010	100	011	001	0000	01	1	0	0	0	0
122	01111000	011	100	000	0000	00	0	1	0	0	1
170	01111001	010	000	001	0000	01	0	1	0	0	0
171	01111010	010	000	100	0000	00	0	0	0	0	0
172	01111011	101	011	000	0000	00	0	0	0	0	0
173	00000000	011	101	000	0000	00	0	0	0	0	0

Рис. 7.24. Реализация микропрограммы, приведенной на рис. 7.21, с использованием поля адреса следующей микрокоманды (коды сигналов указаны на рис. 7.23)

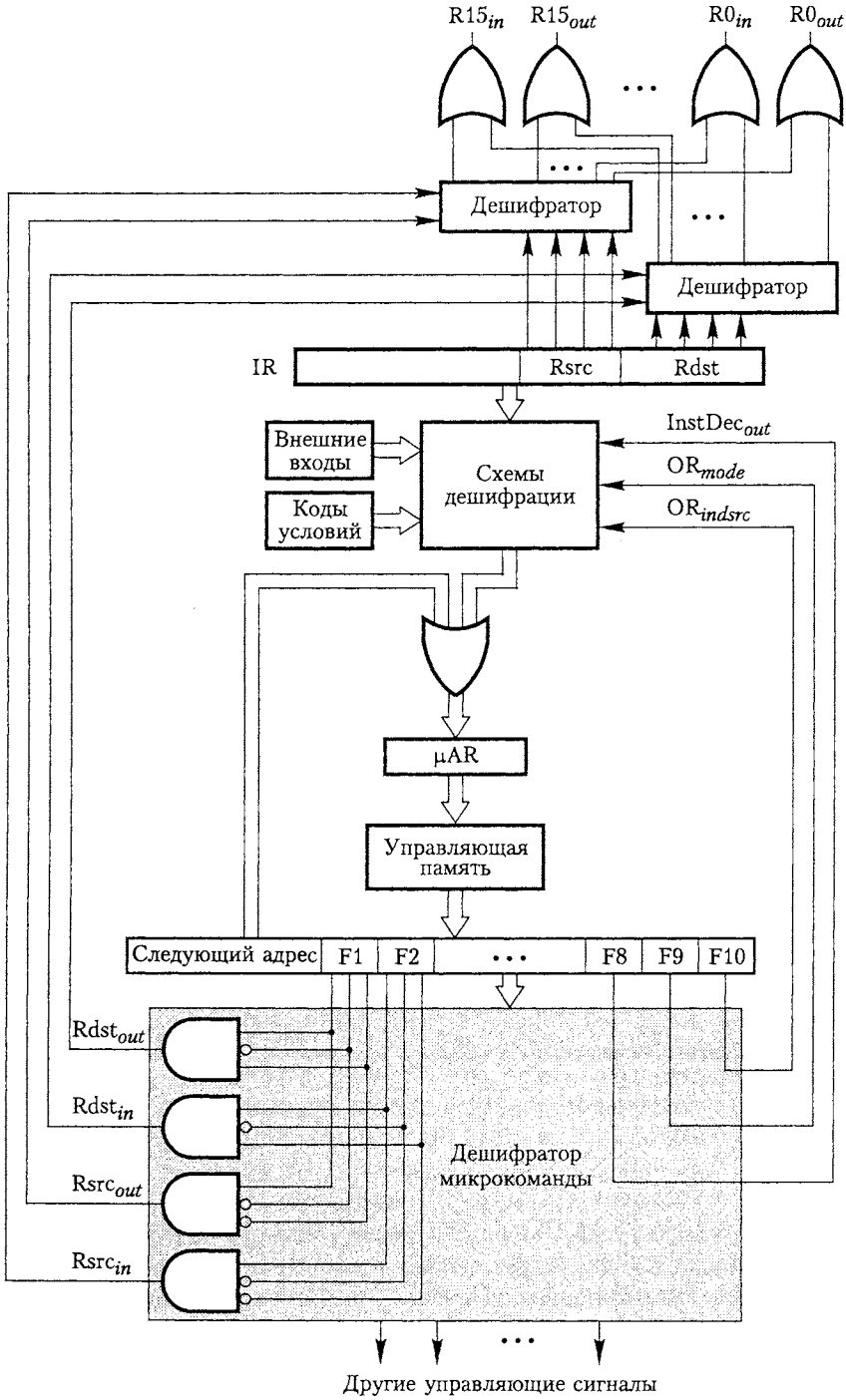


Рис. 7.25. Подробная схема формирования управляющих сигналов

На рис. 7.25 приведена более подробная схема управляющей структуры, представленной на рис. 7.22. Здесь показано, как дешифруются управляющие сигналы из полей микрокоманды и как они используются при управлении последовательностью выполнения микрокоманд. Детальная схема логического сложения разрядов представлена на рис. 7.26.

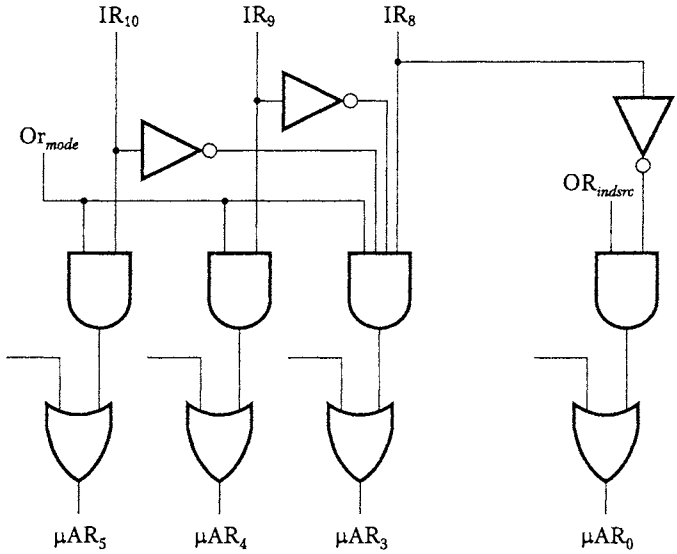


Рис. 7.26. Управляющая схема логического сложения разрядов (часть схемы дешифрации, показанной на рис. 7.25)

7.5.5. Упреждающая выборка микрокоманд

Одним из недостатков микропрограммного управления является замедление работы компьютера вследствие задержек на выборку микрокоманд из управляющей памяти. Но этот процесс ускоряется, если очередная микрокоманда выбирается еще до завершения работы предыдущей микрокоманды, так как время выполнения и время выборки микрокоманд накладываются друг на друга.

С упреждающей выборкой микрокоманд связаны некоторые организационные трудности. Часто для определения адреса следующей микрокоманды нужны флаги состояния и результаты выполнения текущей микрокоманды. В подобном случае из управляющей памяти может быть выбрана не та микрокоманда. Если такое случится, можно просто повторить выборку, на этот раз с правильным адресом, что, конечно же, усложнит аппаратную реализацию процесса. Однако по сравнению с преимуществами упреждающей выборки эти недостатки незначительны, поэтому данная технология часто используется в современных компьютерах.

7.5.6. Эмуляция

Микропрограммное управление обеспечивает простоту, гибкость и относительно низкую стоимость выполнения машинных команд. К тому же наряду с этими

важными достоинствами у него имеется ряд интереснейших возможностей. Его гибкость в использовании ресурсов компьютера позволяет применять самые разные типы команд. Имея компьютер с некоторым набором команд, можно определить дополнительные машинные команды и реализовать их в виде дополнительных микропрограмм.

Развитие этой идеи открывает еще одну интересную возможность. Предположим, что мы добавили к системе команд компьютера M1 совершенно новый набор команд — систему команд другого компьютера, M2. Программы, написанные на машинном языке компьютера M2, можно выполнять на компьютере M1 — получается, что M1 *эмулирует* компьютер M2. Эмуляция позволяет заменять устаревшее оборудование более современным. И если новый компьютер полностью эмулирует исходный, то никакие изменения в существующие программы вносить не требуется. Благодаря эмуляции старого оборудования переход на новую технику производится с минимальными затратами.

Если компьютеры имеют сходную архитектуру, эмуляция одного из них другим не представляет труда. Однако эмулировать можно и компьютеры с совершенно разной архитектурой.

7.6. Резюме

В этой главе рассматривалась внутренняя организация центрального процессорного устройства компьютера. В современных машинах применяются самые разные вариации описанной здесь архитектуры. При выборе конкретного решения конструкторы стараются найти оптимальное соотношение между скоростью выполнения программ и стоимостью реализации процессора. Кроме того, учитываются особенности выбранной технологии, гибкость системы с точки зрения модификации и потребность в специальных возможностях системы команд компьютера.

Мы рассмотрели два подхода к реализации управляющего блока процессора — аппаратное управление и микропрограммное управление. Аппаратное решение выбирают в том случае, если на первом плане стоит скорость функционирования компьютера. Если же более важным критерием является гибкость реализации системы команд, то предпочтение отдают микропрограммному управлению.

Упражнения

7.1. Почему чтению информации из памяти и ее записи в память предшествует этап, в течение которого система ожидает завершения операции с памятью WMFC (Wait for Memory Function Completed)?

7.2. В процессоре выполняется управляющая последовательность, подобная приведенной на рис. 7.6. Предположим, что операции чтения из памяти и записи в память занимают столько же времени, сколько один временной цикл процессора, и при этом процессор и память управляются одним и тем же тактовым сигналом. Оцените время выполнения данной последовательности.

7.3. Выполните упражнение 7.2 для компьютера, в котором время доступа к памяти равно двум тактам процессора.

7.4. Предположим, что на рис. 7.1 задержка на распространение сигнала по шине и через АЛУ составляет 0,3 и 2 нс соответственно. Время установки регистров равно 0,2 нс, а время удержания — 0. Каков минимальный такт?

7.5. Напишите последовательность управляющих действий, необходимых для выполнения каждой из перечисленных далее команд, если процессор имеет такую структуру, как показано на рис. 7.1.

а) Прибавить непосредственно заданное число NUM к содержимому регистра R1.

б) Прибавить содержимое памяти по адресу NUM к содержимому регистра R1.

в) Прибавить к содержимому регистра R1 содержимое памяти, расположенной по адресу, который, в свою очередь, находится по адресу NUM.

Предполагается, что каждая команда состоит из двух слов. Первое из них определяет операцию и режим адресации, а второе содержит число NUM.

7.6. Три команды в упражнении 7.5 могут включать одинаковые управляющие действия. Однако некоторые из этих действий могут выполняться в другом порядке. Разработайте схему осуществления этих общих действий с целью упрощения блока шифрации, показанного на рис. 7.11.

7.7. Рассмотрим команду сложения, управляющая последовательность которой приведена на рис. 7.6. Процессор управляется постоянно работающим генератором тактовых сигналов, и каждый его такт длится 2 нс. Как долго процессор должен ждать на шаге 2 и 5, если считать, что операция чтения из памяти занимает 16 нс? В течение какого времени (в процентах от всего времени выполнения этой команды) процессор бездействует?

7.8. 32-разрядная машина с побайтово адресуемой памятью поддерживает автоинкрементный и автодекрементный режимы адресации. В этих режимах содержимое регистра адреса либо увеличивается, либо уменьшается на 1, 2 или 4, в зависимости от длины операнда. Как можно изменить схему, показанную на рис. 7.1, с тем чтобы упростить эти операции.

7.9. Приведите возможную управляющую последовательность для реализации команды

MUL R1,R2

для процессора, схема которого изображена на рис. 7.1. Эта команда перемножает содержимое регистров R1 и R2 и помещает результат в регистр R2. Если в произведении имеются старшие разряды, они отбрасываются. Какие дополнительные управляющие сигналы нужны для выполнения этой последовательности, если умножитель организован так, как показано на рис. 6.7?

7.10. Приведите управляющую последовательность для команды Branch on Negative (переход, если отрицательно) при условии, что процессор имеет структуру, подобную представленной на рис. 7.8.

7.11. Приведите управляющую последовательность для команды Branch to Subroutine (переход к подпрограмме) одного из процессоров, описанных в главе 3. Предполагается, что процессор имеет структуру, подобную представленной на рис. 7.1.

7.12. Выполните упражнение 7.11 для процессора, структура которого показана на рис. 7.8.

7.13. На рис. 7.3 вы видите управляемый фронтом сигнала триггер, используемый для реализации регистров процессора. Рассмотрите операцию пересылки данных из одного регистра в другой. Подробно проанализируйте процесс тактирования этой операции и опишите потенциальные трудности, связанные с заменой указанного триггера простой вентильной защелкой, изображенной на рис. А.27.

7.14. Использование мультиплексора и обратного соединения в схеме, приведенной на рис. 7.3, позволяет обойтись на тактовом входе без вентиля, предназначенного для открытия и закрытия входа регистра. Воспользовавшись временной диаграммой, объясните, какие проблемы могут возникнуть в случае применения вентильного соединения.

7.15. Предположим, что регистровый файл на рис. 7.8 реализован как память с произвольным доступом (RAM), к которой в любой момент времени можно обратиться либо для чтения, либо для записи. В операции R1 ← [R1] + [R2] регистр R1 является и источником, и приемником данных. Как следует применять дополнительные защелки на входе или на выходе RAM, чтобы получить возможность работать с файлом в режиме главный—подчиненный. С помощью временной диаграммы поясните, как новая структура позволяет использовать регистр R1 и в роли источника, и в роли приемника данных во время одного и того же такта.

7.16. Сигнал Run на рис. 7.11 устанавливается в 0, чтобы приостановить приращение значения счетчика шагов на время, пока выполняется операция чтения из памяти или записи в память. Проанализируйте временную диаграмму, показанную на рис. 7.5, и создайте диаграмму состояний для управляющей схемы, генерирующей этот сигнал. Начертите соответствующую схему.

7.17. Управляющий сигнал MDR_{inE} подается вслед за тактом, на котором устанавливается сигнал Read, и снимается по завершении пересылки данных между памятью и процессором (рис. 7.5). Разработайте управляющую схему, генерирующую сигнал MDR_{inE}.

7.18. Рассмотрите 16-разрядную машину с побайтовой адресацией и организацией, показанной на рис. 7.1. Байты с четными и нечетными адресами пересылаются по восьми старшим и восьми младшим адресным линиям шины памяти. Приведите подходящую вентильную схему для соединения регистра MDR с шиной памяти и внутренней шиной процессора, позволяющую выполнять пересылку отдельных байтов. В процессоре обрабатываемый байт должен всегда находиться в младшем байте регистра.

7.19. Разработайте генератор частоты, используя инвертор в качестве элемента, обеспечивающего задержку. Какой будет частота генератора, если задержка в инверторе равна T?

Модифицируйте созданный генератор таким образом, чтобы выдача сигналов начиналась и прекращалась под управлением сигнала, подаваемого на асинхронный вход RUN. Когда генератор остановлен, ширина последнего импульса на его выходе должна быть равной T, вне зависимости от того, в какой момент времени будет снят сигнал RUN.

7.20. Некоторые управляющие шаги в процессоре выполняются дольше остальных. Тактовый сигнал должен задаваться сигналом Long/Short (длинный/короткий) таким образом, чтобы при установке последнего в 1 длительность управляющего шага удваивалась. Предположим, что у счетчика управляющих шагов имеется вход Enable и что на положительном фронте сигнала, подаваемого на этот вход, при условии Enable = 1 выполняется приращение значения данного счетчика. Разработайте схему, генерирующую сигнал Enable с целью изменения длины управляющих шагов.

7.21. Выход сдвигового регистра инвертируется и подается на его вход, образуя схему, которая называется счетчиком Джонсона.

а) Какова последовательность счета в 4-разрядном счетчике Джонсона начиная с состояния 0000?

б) Объясните, как использовать счетчик Джонсона для выдачи показанных на рис. 7.11 тактовых сигналов T1, T2 и т. д. при условии, что всего должно быть не более 10 временных интервалов.

7.22. В АЛУ процессора для выполнения операций сдвига и циклического сдвига применяется сдвиговый регистр, показанный на рис. У7.1.

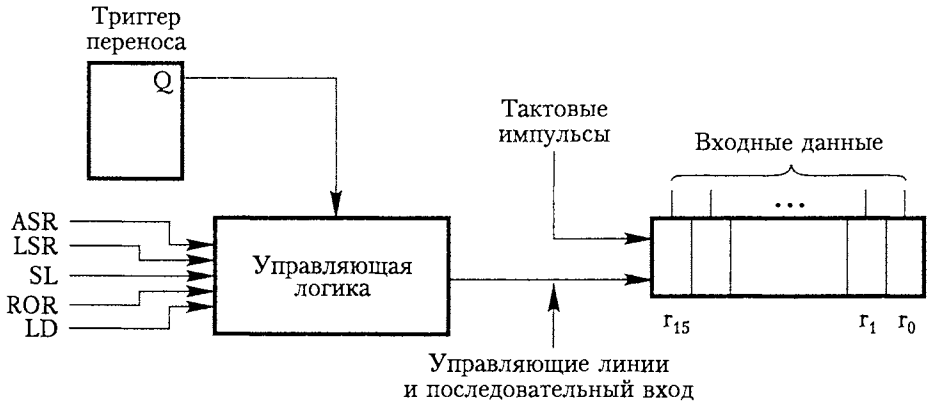


Рис. У7.1. Организация управления сдвиговым регистром для упражнения 7.22

Управляющий логический блок этого регистра имеет пять входов:

ASR — арифметический сдвиг вправо;
LSR — логический сдвиг вправо;
SL — сдвиг влево;
ROR — циклический сдвиг вправо;
LD — параллельная загрузка.

Все операции сдвига и загрузки управляются одним тактовым входом. Сдвиговый регистр реализован в виде D-триггера, управляемого фронтом сигнала. Приведите полную логическую схему управляющей логики и разрядов сдвигового регистра r_0, r_1 и r_{15}.

7.23. У представленного на рис. У7.2 цифрового контроллера имеется три выхода (X, Y, Z) и два входа (A, B). Управляется он извне тактовым сигналом. В контроллере происходит одна и та же последовательность событий. В начале первого такта сигнал на выходе X устанавливается в 1. В начале второго такта в 1 устанавливается сигнал на выходе Y или Z, что зависит от значения сигнала на входе A на предыдущем такте — 1 или 0 соответственно. Далее контроллер ожидает установки в 1 сигнала B. На следующем положительном фронте сигнала контроллер устанавливает в 1 выход Z, на время одного такта, а затем на время одного такта он сбрасывает все сигналы в 0. Начиная с положительного фронта тактового сигнала эта последовательность повторяется. Нарисуйте диаграмму состояний и приведите для этого контроллера подходящую логическую схему.

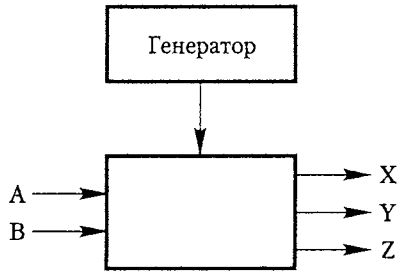


Рис. У7.2. Цифровой контроллер для упражнения 7.23

7.24. Напишите микропрограмму, такую как на рис. 7.21, для команды

MOV X(Rsrc),Rdst

Исходный и результирующий операнды должны быть заданы соответственно в режиме индексной и регистровой адресации.

7.25. Условием перехода машинной команды BGT (Branch if > 0) является выражение Z + (N ⊕ V) = 0, где Z, N и V — флаги условий «нуль», «отрицательное значение» и «переполнение». Напишите микропрограмму, реализующую данную команду. Приведите схему для тестирования кодов условий.

7.26. Напишите объединенную микропрограмму, реализующую команды BGT (Branch if > 0 — переход если больше нуля), BPL (Branch if Plus — переход если плюс) и BR (Branch Unconditionally — безусловный переход). Условиями перехода для команд BGT и BPL являются Z + (N ⊕ V) = 0 и N = 0. Сколько микрокоманд будет содержать эта программа? Сколько микрокоманд потребовалось бы, если бы для каждой машинной команды использовалась отдельная микропрограмма?

7.27. На рис. 7.21 вы видите пример микропрограммы, в которой для модификации адресов микрокоманд применяется логическое сложение разрядов.

Напишите эквивалентную программу без использования логического сложения разрядов, но с командами условного перехода. Сколько для нее потребуется дополнительных микрокоманд? Предполагается, что микрокоманды условного перехода могут проверять некоторые разряды регистра IR.

7.28. Покажите, как можно модифицировать микропрограмму, приведенную на рис. 7.20, с тем чтобы реализовать команду микропроцессора 68000

Add src,Rdst

7.29. Как модифицировать программу, приведенную на рис. 7.20, чтобы реализовать универсальную команду

MOVE src,dst

в которой исходный и результирующий операнды могут задаваться с использованием любого из пяти указанных режимов адресации.

7.30. На рис. У7.3 приведена часть последовательности микрокоманд, соответствующая одной из машинных команд микропрограммируемого компьютера. За микрокомандой B следует микрокоманда C, E, F или I, что зависит от значений разрядов b_6 и b_5 в регистре машинной команды. Сравните три описанные ниже реализации.

а) Последовательность микрокоманд управляется счетчиком микропрограммы. Переходы выполняются с помощью микрокоманд такого вида:

if b_6b_5 branch to X

где b_6b_5 — это условие перехода, а X — адрес перехода.

б) Последовательность микрокоманд, как и выше, управляется счетчиком микропрограммы, но микрокоманда перехода имеет вид

branch to X, OR

где X — это базовый адрес перехода. Адрес перехода модифицируется посредством логического сложения разрядов b_5 и b_6 с соответствующими разрядами X.

в) В каждой микрокоманде имеется поле, где задается адрес следующей микрокоманды, которая поддерживает логическое сложение разрядов.

Назначьте всем микрокомандам на рис. У7.3 адреса, которые позволили бы реализовать все перечисленные выше задания. В некоторых случаях вам могут потребоваться команды перехода. Адреса можно выбирать произвольно — лишь бы они согласовывались с используемым методом управления последовательностью выполнения микрокоманд. Например, для первого задания можно выбрать адреса, перечисленные ниже.

Адрес	Микрокоманда
00010	A
00011	B
00100	if b_6b_5 = 00 переход к XXXXX
...	...
XXXXX	C

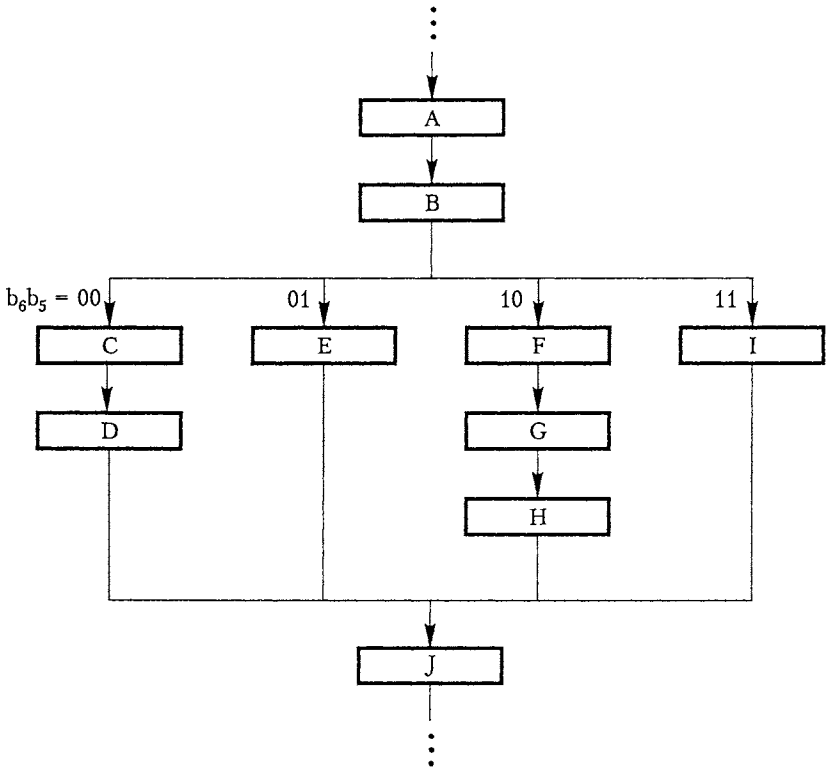


Рис. У7.3. Последовательность выполнения микрокоманд в упражнении 7.30

7.31. Требуется сократить количество битов, необходимых для кодирования управляющих сигналов в схеме, показанной на рис. 7.19. Предложите новую схему кодирования, сокращающую число битов на два. Как это отразится на количестве управляющих шагов, необходимых для реализации команды?

7.32. Предложите новую схему кодирования управляющих сигналов для схемы, показанной на рис. 7.19, которая бы сократила число битов микрокоманды до 12. Покажите результат ее действия на примере управляющих последовательностей, приведенных на рис. 7.6 и 7.7.

7.33. Разработайте формат микрокоманд, подобный приведенному на рис. 7.19, для процессора, который организован по принципу, продемонстрированному на рис. 7.8.

7.34. Каковы преимущества горизонтального и вертикального форматов команд? Свяжите ваш ответ с ответами к упражнениям 7.31 и 7.32.

7.35. Каковы преимущества и недостатки аппаратного и микропрограммного управления?

Глава 8

Конвейерная обработка команд

- Конвейерная обработка как способ параллельного выполнения машинных команд
- Конфликты, снижающие производительность процессоров с конвейерной организацией, и способы ликвидации их последствий
- Аппаратные и программные средства поддержки конвейерной обработки команд
- Учет конвейерной обработки при формировании системы команд
- Суперскалярные процессоры

В предыдущих главах вы познакомились с основными компонентами компьютера. Настоящая глава посвящена конвейерной обработке команд, цель которой — повышение производительности современных компьютеров. В первую очередь рассматриваются основные принципы конвейерной обработки и ее влияние на производительность компьютера. После этого дается обзор машинных команд, предназначенных для поддержки данного процесса. Мы проследим, как отражается на производительности выбор той или иной команды, а также последовательность их активизации. Для поддержки конвейерной обработки используются сложные технологии компиляции и так называемые *оптимизирующие компиляторы*. Наряду с решением других задач эти компиляторы реорганизуют последовательность выполняемых программой операций с целью максимального повышения эффективности конвейерной обработки.

8.1. Базовые концепции

Скорость выполнения программ зависит от многих факторов. Одним из способов ее повышения является внедрение передовых технологий при проектировании и изготовлении процессоров и модулей основной памяти. Эти технологии могут быть направлены как на повышение быстродействия функциональных устройств, так и на совершенствование их архитектуры с целью увеличения количества параллельно выполняемых операций. Итак, второй метод предполагает увеличение количества операций, выполняемых за одну секунду, при этом время, отводимое одной операции, остается неизменным.

В предыдущих главах неоднократно затрагивалась тема параллельного выполнения различных операций. В главе 1 была представлена концепция мультипрограммирования и объяснялось, как обмен данными с устройствами ввода-вывода может выполняться одновременно с вычислительными операциями. Отметим, что это возможно благодаря устройствам, поддерживающим технологию ПДП, поскольку после того как процессор инициирует операцию ввода-вывода, такое устройство сможет продолжить ее осуществление самостоятельно.

Особенно эффективным способом организации параллельных операций в компьютерной системе является конвейерная обработка команд. Ее основная идея очень проста. Понятие конвейера нам хорошо знакомо из жизни — это сборочная линия на фабрике или заводе, на которой последовательно выполняется ряд манипуляций. Представим, как, например, собирается автомобиль. Предположим, на первом рабочем месте сборочной линии изготавливается шасси, на втором добавляется корпус, на третьем устанавливается мотор и т. д. Пока первая группа рабочих монтирует мотор одной из машин, вторая закрепляет корпус другой машины, а третья готовит новое шасси для еще одной машины. Работа над каждой машиной может продолжаться несколько дней, но при этом каждые несколько минут с конвейера сходит готовый автомобиль.

Теперь давайте посмотрим, каким образом эта идея воплощена в компьютерном мире. Процессор выполняет программу, по очереди выбирая из памяти и активизируя ее команды. Обозначим шаги выборки и выполнения команды I_i как F_i и E_i. Процесс выполнения программы представляет собой последовательность шагов выборки и активизации команд, как показано на рис. 8.1, *а*.

Теперь рассмотрим компьютер с двумя отдельными функциональными блоками — для выборки команд и для их выполнения (рис. 8.1, *б*). Команда извлекается из памяти устройством выборки и помещается в промежуточный буфер B1, который нужен для того, чтобы блок выполнения мог обрабатывать команду в то время, как блок выборки уже берет из памяти следующую. Результаты выполнения команды размещаются по указанному в ней адресу. Мы предполагаем, что источник и приемник данных находятся в блоке, помеченном как «Блок выполнения».

Этот компьютер управляется тактовым сигналом с такой частотой, при которой и шаг выборки, и шаг выполнения занимают один такт. Схематически процесс работы компьютера представлен на рис. 8.1, *в*. Во время первого такта блок выборки извлекает из памяти команду I_1 (шаг F_1) и сохраняет ее в буфере B1. На втором такте блок выборки берет из памяти команду I_2 (шаг F_2). Тем временем блок выполнения осуществляет операцию, указанную в команде I_1, которую он считывает из буфера B1 (шаг E_1). По окончании второго такта обработка команды I_1 завершается, к этому моменту из памяти уже считывается команда I_2. Теперь в буфере B1 сохраняется команда I_2, заменяя команду I_1, которая больше не нужна. Шаг E_2 производится блоком выполнения в течение третьего такта, пока команда I_3 извлекается из памяти блоком выборки. И так далее. В результате и блок выборки и блок выполнения команд все время заняты, а скорость команд вдвое больше, чем при последовательной обработке, которая схематически показана на рис. 8.1, *а*. Блоки выборки и выполнения команд, показанные на рис. 8.1, *б*, составляют двухступенчатый конвейер, на каждой ступени которого совершается

один шаг обработки команды. Для хранения информации, передаваемой с одной ступени обработки команды на другую, применяется промежуточный буфер B1. В конце каждого такта в этот буфер загружается новая информация.

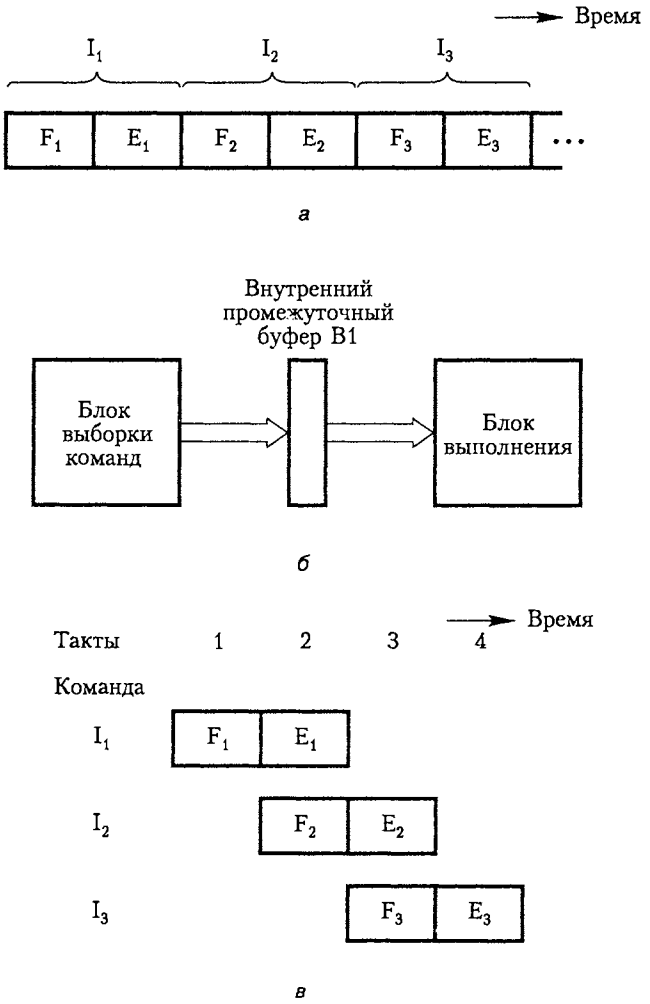


Рис. 8.1. Принцип конвейерной обработки команд: последовательное выполнение (*а*); аппаратная организация (*б*); конвейерная обработка команд (*в*)

Процесс обработки команды может быть разбит на количество шагов, превышающее два. Например, конвейерный процессор способен обрабатывать команды за четыре шага. Рассмотрим их.

F: Выборка (Fetch) — чтение команды из памяти.

D: Декодирование (Decode) — декодирование команды и выборка ее исходных операндов.

E: Выполнение (Execute) — выполнение заданной в команде операции.

W: Запись (Write) — сохранение результата по целевому адресу.

В какой последовательности происходят события при такой схеме обработки команды, показано на рис. 8.2, *а*. В каждый момент процессор обрабатывает четыре команды. Это означает, что он содержит четыре отдельных функциональных блока (рис. 8.2, *б*).

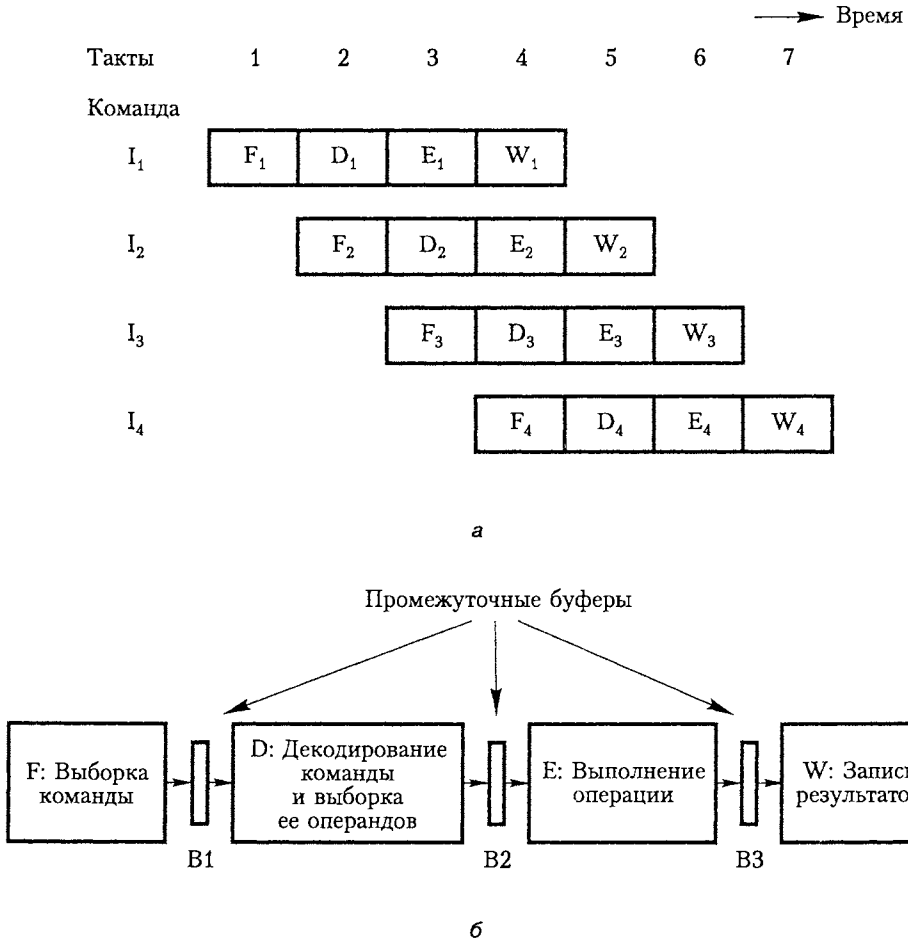


Рис. 8.2. Структура 4-ступенчатого конвейера: выполнение команды за 4 шага (*а*); аппаратная организация (*б*)

Блоки должны решать свои задачи параллельно, не мешая друг другу. Информация передается от одного блока к другому через промежуточный буфер. По мере обработки команды конвейером в промежуточные буферы должна поступать вся необходимая для очередной ступени информация. Вот какую информацию должен содержать буфер для четвертого такта.

- Буфер B1: команда I_3, выбранная на такте 3 и обработанная блоком декодирования команды.
- Буфер B2: исходные операнды команды I_2 и спецификация выполняемой операции. Эта информация сформирована схемами дешифратора на такте 3. Буфер B2 включает также информацию, необходимую для шага W_2, на котором осуществляется запись команды I_2. Хотя она не нужна на ступени выполнения, на следующем такте она передается на ступень записи для осуществления соответствующей операции.
- Буфер B3: результаты, которые сгенерированы блоком выполнения команды, и информация о месте назначения данных команды I_1.

8.1.1. Роль кэш-памяти

Каждая ступень конвейерного выполнения команды должна завершаться за один такт. Это условие соблюдается, если такт достаточно велик. В случае, когда блокам конвейера для решения задач требуется разное время, такт равен времени выполнения самой продолжительной задачи. Блок, работа которого завершилась раньше, «бездействует» до конца такта. Поэтому конвейеризация наиболее эффективна в том случае, когда для решения стоящих на разных ступенях задач необходимо примерно одинаковое время.

Особое значение это представляет для шага выборки команды, которому на рис. 8.2, *а* назначен один такт. Продолжительность такта должна быть достаточной для полного выполнения шага. Однако время доступа к основной памяти может в десять раз превышать время выполнения одной конвейерной ступени внутри процессора (например, на которой суммируются два числа). Поэтому в том случае, когда для выборки каждой команды требуется обращение к основной памяти, конвейерная обработка команд бесполезна.

Проблема доступа к памяти решается с помощью кэш-памяти. Если кэш находится на той же микросхеме, что и процессор, для доступа к ней требуется то же время, что и для выполнения других базовых операций внутри процессора. Поэтому только при наличии кэша становится возможным разделение процессов выборки и обработки команд на относительно равные шаги, которые можно осуществлять на разных ступенях конвейера. При этом продолжительность такта выбирается в соответствии с тем временем, в течение которого выполняется самый продолжительный шаг.

8.1.2. Производительность конвейерной обработки команд

Конвейерный процессор на рис. 8.2 обрабатывает команду за один такт, благодаря чему он выполняет программы в четыре раза быстрее, чем обычный последовательный процессор. Повышение производительности, которое может быть получено при реализации конвейерной обработки, прямо пропорционально количеству ступеней конвейера. Однако это значение может быть достигнуто только в том случае, если команды программы выполняются безо всяких задержек. Но на практике так не бывает.

В силу различных причин одной из ступеней конвейерной обработки может не хватить выделенного ей диапазона времени. Например, в 4-ступенчатом конвейере, изображенном на рис. 8.2, *б*, на ступени E, которой выделен один такт, выполняется арифметическая или логическая операция. Для большинства операций этого вполне достаточно, но некоторым, в частности делению, требуется больше времени. Рассмотрим пример, который проиллюстрирован на рис. 8.3. Для совершения операции, задаваемой командой I_2, нужно три такта: от 4 до 6. Поэтому выполнение операции записи на тактах 5 и 6 откладывается — данные, которые должны быть обработаны этой командой, еще не готовы. Информация в буфере B2 должна оставаться неприкосновенной, пока не завершатся операции, производимые на ступени выполнения. Это означает, что ступень 2, а вместе с ней и ступень 1, блокируются до конца операции и не могут принимать новые команды, поскольку нельзя перезаписать информацию в буфере B1. Таким образом, как показано на рисунке, выполнение шагов D_4 и F_5 должно быть отложено.

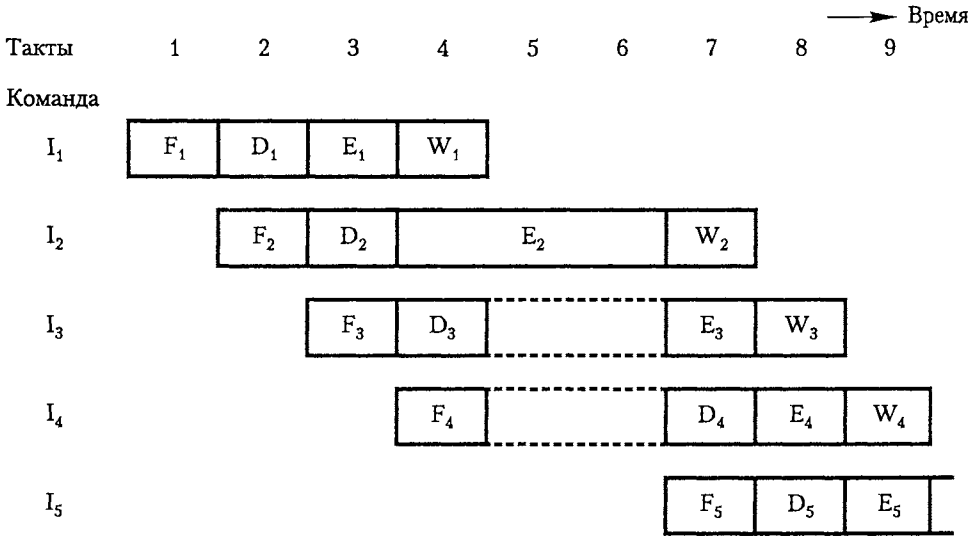
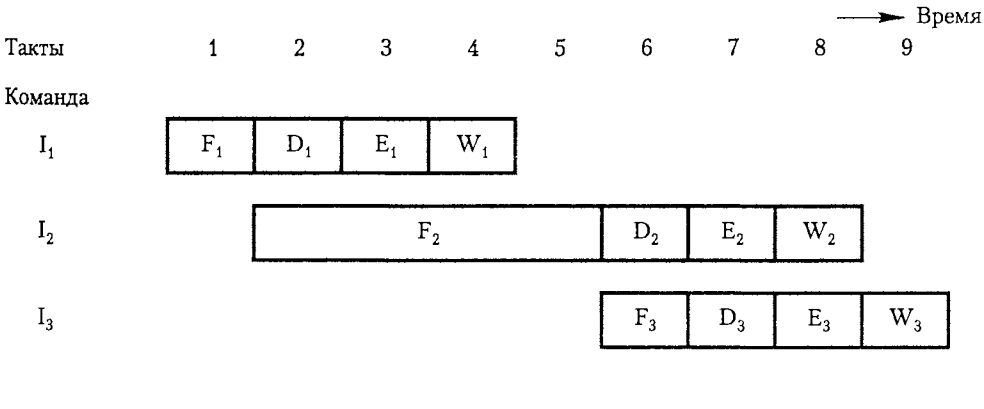


Рис. 8.3. Результаты выполнения операции, занимающей более одного такта

Работа конвейера, который вы видите на рис. 8.3, *приостанавливается* на два такта. Его нормальное функционирование возобновляется во время такта 7. Обстоятельства, в силу которых приостанавливается работа конвейера, называют *конфликтами*. Мы рассмотрели пример конфликта по данным. Это ситуация, когда либо исходный, либо результирующий операнд команды не доступен в положенное время. В результате операция, которой он нужен, откладывается, а работа конвейера приостанавливается.

Еще одна причина останова конвейера может заключаться в задержке поступления очередной команды. Возможная причина такой задержки — промах при попытке выборки команды из кэш-памяти. Конфликты этого типа называются *конфликтами по управлению*. На рис. 8.4 показано, как отсутствие команды в кэше

отражается на работе конвейера. Команда I_1 выбирается из кэша во время такта 1 и выполняется обычным образом. Далее, на такте 2, производится выборка команды I_2, а при обращении к кэшу происходит промах. Из-за этого работа блока выборки команд приостанавливается до получения команды I_2. Мы полагаем, что команда I_2 поступает и загружается в буфер B1 в конце такта 5. С этого момента возобновляется нормальная работа конвейера.



а

Время

Такты	1	2	3	4	5	6	7	8	9
Ступень									
F: Выборка	F_1	F_2	F_2	F_2	F_2	F_3			
D: Декодирование		D_1	*	*	*	D_2	D_3		
E: Выполнение			E_1	*	*	*	E_2	E_3	
W: Запись				W_1	*	*	*	W_2	W_3

* Простой конвейера

б

Рис. 8.4. Останов конвейера, вызванный промахом при обращении к кэшу на шаге F_2: шаги при выполнении команды на последовательных тактах (*а*); действия, выполняемые на ступенях конвейера во время последовательных тактов (*б*)

Рис. 8.4, *б* — это альтернативное изображение происходящего в конвейере в случае промаха при обращении к кэш-памяти. Показано, какие операции выполняются во время каждого такта на ступенях конвейера. Обратите внимание: во время тактов 3–5 простаивает блок декодирования, во время тактов 4–6 — блок выполнения, а во время тактов 5–7 — блок записи. Периоды простоя иногда называют *остановом конвейера* или *пузырями*. Образовавшись в результате задержки на одной ступени, пузырь спускается вниз, пока не достигнет последнего блока.

Различают также *структурные конфликты*. Они возникают, когда двум командам требуется одновременный доступ к аппаратному ресурсу. Структурные конфликты наиболее вероятны при обращении к памяти. Предположим, что одной из находящихся в конвейере команд требуется доступ к памяти на ступени выполнения или записи, а другой — на ступени выборки. Если команды и данные расположены в одном и том же кэше, они не могут извлекаться из кэша одновременно. Поэтому пока одна команда не получит из кэша необходимую информацию, другая не сможет продолжить работу. Во избежание таких задержек многие процессоры оборудуются раздельными кэшами команд и данных.

Пример структурного конфликта приведен на рис. 8.5. Показано, как на 4-ступенчатом конвейере выполняется команда

Load X(R1),R2

На шаге E_2 такта 4 вычисляется адрес памяти X+[R1], по которому происходит обращение к памяти на такте 5. Прочитанный из памяти операнд на такте 6 записывается в регистр R2. Это означает, что шаг выполнения этой команды занимает два такта (4 и 5). В результате конвейер приостанавливается на один такт, поскольку обеим командам, I_2 и I_3, на такте 6 требуется доступ к регистровому файлу. И хотя в наличии имеются и команды и данные, конвейер приостанавливается из-за того, что единственный аппаратный ресурс, регистровый файл, не способен одновременно обрабатывать две операции. Если бы у регистрового файла было два входных порта, он смог бы обработать за один прием две операции записи, и конвейер бы не остановился. В общем случае для предотвращения структурных конфликтов нужно, чтобы в микросхеме процессора было достаточное количество аппаратных ресурсов.

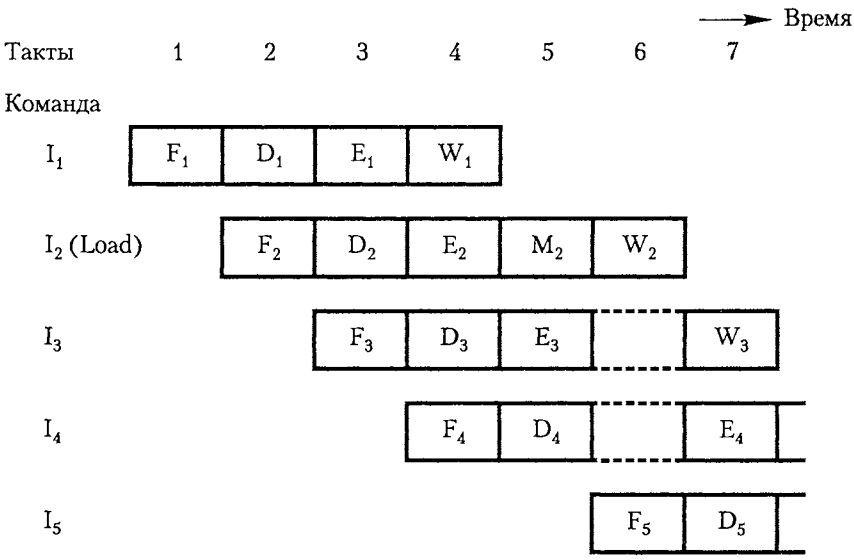


Рис. 8.5. Воздействие команды загрузки на работу конвейера

Важно понимать, что конвейерная обработка команд не приводит к ускорению выполнения каждой отдельной команды. Скорость команд остается прежней, но зато на единицу времени приходится большее их количество. При этом, как только на одной из ступеней конвейера возникает затор и работа не завершается за один такт, весь конвейер останавливается в ожидании завершения процесса. Если описанные ситуации возникают достаточно часто, производительность компьютера снижается.

Важнейшей задачей разработчиков процессоров является выявление конфликтов, которые могут привести к приостановке процессора, и поиск путей их устранения. Следующие разделы содержат подробную информацию о конфликтах различных типов, а также технологиях, позволяющих снизить их отрицательное воздействие на производительность. К вопросу производительности мы вернемся в разделе 8.8.

8.2. Конфликты по данным

Конфликт по данным — это ситуация, когда конвейер останавливается из-за отсутствия данных, над которыми должна производиться очередная операция (рис. 8.3). Тема доступности данных требует отдельного обсуждения.

Рассмотрим программу, содержащую команды I_1 и I_2, которые обрабатываются одна за другой. Когда эта программа выполняется конвейером, активизация команды I_2 может произойти до завершения действия, заданного командой I_1. Это означает, что команде I_2 еще не доступны результаты, генерируемые командой I_1. В рассматриваемом случае результаты выполнения команд конвейерным процессором и последовательного выполнения должны быть идентичны. Возможность получения неправильных результатов при параллельном выполнении команд можно продемонстрировать на очень простом примере. Предположим, что A = 5. Рассмотрим следующие две операции:

$$A \leftarrow 3 + A$$
$$B \leftarrow 4 \times A$$

Если операции выполняются по порядку, результат B равен 32. Однако в параллельном режиме при вычислении значения B используется исходное значение A, то есть 5, и результат получается неверным. Команды программы должны выполнять эти операции одну за другой, поскольку данные, применяемые второй командой, зависят от результатов обработки первой. Следующие команды можно выполнить параллельно, поскольку связанные с ними операции независимы:

$$A \leftarrow 5 \times C$$
$$B \leftarrow 20 + C$$

В этом примере продемонстрировано ограничение, без соблюдения которого невозможно получение правильных результатов. Когда две операции зависят одна от другой, они должны выполняться последовательно и в правильном порядке. У этого очевидного ограничения имеются далеко идущие последствия, от знания

которых зависит понимание альтернативных конструкторских решений и компромиссов, на которые приходится идти при разработке конвейерных процессоров.

Вернемся к конвейеру, изображенному на рис. 8.2. Только что описанная зависимость команд имеет место тогда, когда адрес результирующего операнда одной команды является адресом исходного операнда другой команды. Такие операнды присутствуют в представленных ниже командах:

```
Mul   R2,R3,R4
Add   R5,R4,R6
```

Результат умножения помещается в регистр R4, который является одним из исходных операндов команды сложения. Если операция умножения завершается за один такт, выполнение команд происходит так, как показано на рис. 8.6. Во время такта 3 блок декодирования обрабатывает команду сложения, при этом обнаруживается, что ее операндом является регистр R4. Поэтому шаг D данной команды не может быть завершен до тех пор, пока не подойдет к концу шаг W команды умножения. Завершение шага D_2 придется отложить до такта 5, где он обозначен как D_{2A}. Выбор команды I_3 осуществляется на такте 3, но ее декодирование тоже откладывается, поскольку шаг D_3 не может предшествовать шагу D_2. Таким образом, работа конвейера приостанавливается на два такта.

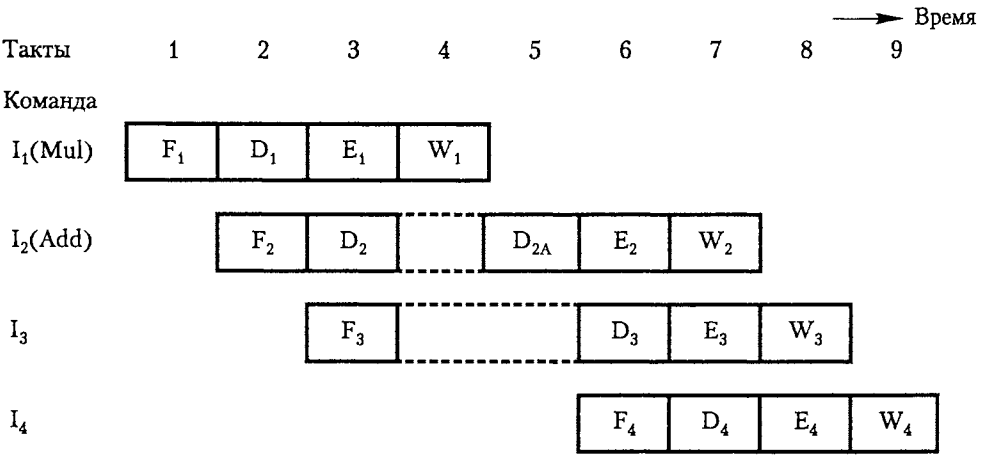
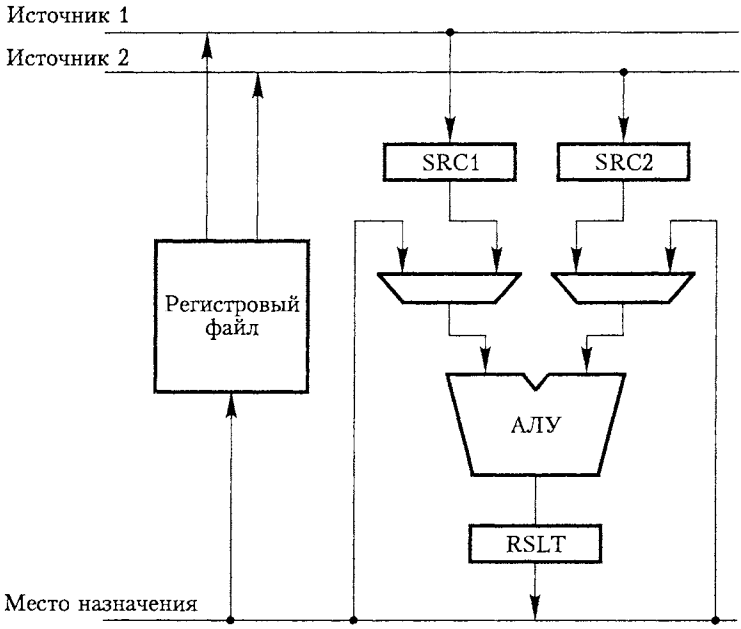


Рис. 8.6. Конвейер приостановлен из-за того, что данные шага D_2 зависят от данных шага W_1

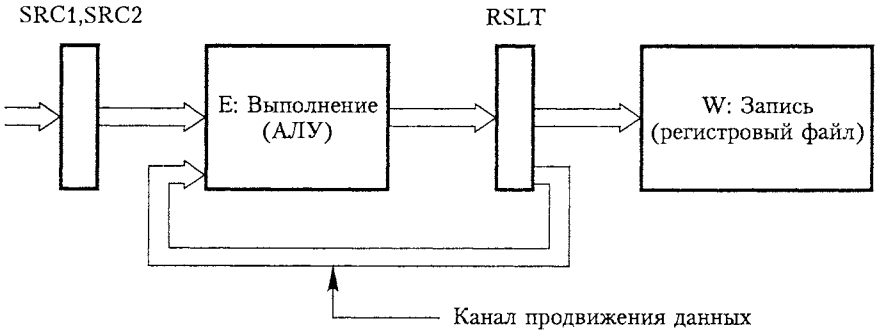
8.2.1. Продвижение операндов

Конфликт, описанный в предыдущем разделе, обусловлен тем, что одна из команд, I_2, ожидает записи данных в регистровый файл. Однако эти данные появляются на выходе АЛУ по завершении шага E_1. Поэтому задержку можно сократить и даже устранить, передав результаты команды I_1 непосредственно команде I_2.

На рис. 8.7, *а* показана часть тракта данных процессора, включающая АЛУ и регистровый файл. Можно провести параллель с трехшинной структурой, представленной на рис. 7.8, однако на рис. 8.7, *а* добавлены регистры SRC1, SRC2 и RSLT. Как показано на рис. 8.7, *б*, эти регистры играют роль промежуточных буферов конвейера. В конвейере, изображенном на рис. 8.2, *б*, регистры SRC1 и SRC2 являются частью буфера B2, а регистр RSLT — частью буфера B3.



а



б

Рис. 8.7. Продвижение операндов в процессоре с конвейерной организацией: канал данных (*а*); исходный и результирующий регистры в конвейере процессора (*б*)

Два мультиплексора, подключенных к входам АЛУ, позволяют считывать не содержимое регистров SRC1 и SRC2, а данные, находящиеся на шине результирующих операндов.

Давайте рассмотрим, как работает конвейер, когда в информационном канале, который показан на рис. 8.7, выполняются команды, приведенные на рис. 8.6. После декодирования команды I_2 и выявления зависимости данных принимается решение об их продвижении. На такте 3 исходный операнд, от которого не зависит следующая команда (регистр R2), считывается и загружается в регистр SRC1. На следующем такте сгенерированное командой I_1 произведение появляется в регистре RSLT; благодаря соединению, предназначенному для продвижения данных, оно может использоваться на шаге E_2. Поэтому выполнение команды I_2 продолжается без остановки.

8.2.2. Программная обработка конфликтов по данным

На рис. 8.6 вам представляется возможность ознакомиться со схемой процесса, в котором зависимость данных выявляется аппаратным путем во время декодирования команды. Если технология продвижения операндов не используется, управляющее аппаратное обеспечение откладывает чтение регистра R4 до такта 5, вследствие чего конвейер приостанавливается на два такта. В качестве альтернативы выявление зависимостей данных и их обработка могут возлагаться на программное обеспечение. В этом случае компилятор сам способен организовать задержку на два такта между командами I_1 и I_2, поместив туда команды NOP (No operation — нет операций).

```
I1:  Mul   R2,R3,R4
     NOP
     NOP
I2:  Add   R5,R4,R6
```

Если ответственность за выявление таких зависимостей всецело возложена на программное обеспечение, для получения правильных результатов компилятор должен вставлять в нужных местах команды NOP. Это свидетельство того, насколько тесна связь между компилятором и аппаратным обеспечением. Многие задачи можно либо решить аппаратно, либо возложить на компилятор. Когда решение той или иной задачи возлагается на компилятор, упрощается аппаратное обеспечение. Более того, компилятор способен реорганизовать программу для поиска наиболее оптимального пути решения задачи. В нашем случае компилятор может произвести реорганизацию команд так, чтобы во время задержки выполнялись не бесполезные команды NOP, а действительно необходимые, что позволит повысить производительность. Если реорганизовать программу не получается, добавление команд NOP приведет к увеличению объема программного кода. Кроме того, нередко встречаются разные аппаратные реализации одной и той же архитектуры процессора, и команды NOP, отвечающие требованиям одной аппаратной реализации, могут не подходить другой.

8.2.3. Побочные эффекты

Зависимости данных, примеры которых приводились в предыдущих разделах, очевидны и легко выявляются, поскольку в двух последовательных командах используется один и тот же регистр, заданный в одной из них как источник, а в другой — как приемник данных. Не исключены ситуации, когда команда изменяет содержимое не того регистра, который задан в ней как приемник данных. Это касается, например, команд, применяющих автоинкрементный или автодекрементный режим адресации. Такая команда не только сохраняет новые данные по целевому адресу, но и изменяет содержимое исходного регистра, используемого для доступа к одному из ее операндов. Поэтому все те действия, которые производятся в случае обнаружения зависимости от результирующих данных, должны совершаться и по отношению к регистрам, для которых выполняется операция автоинкрементации или автодекрементации. Когда команда изменяет данные по адресу, отличному от адреса ее результирующего операнда, говорят, что она производит *побочный эффект*. Например, побочные эффекты производятся операциями со стеком, такими как выталкивание из стека и проталкивание в стек, потому что в них неявно используется автоинкрементная и автодекрементная адресация.

Широко распространен побочный эффект, который состоит в изменении флагов условий, используемых такими командами, как условные переходы и сложение с переносом. Предположим, в регистрах R1 и R2 содержатся целые числа двойной точности, которые мы хотим прибавить к двум другим целым числам двойной точности, хранящимся в регистрах R3 и R4. Решить эту задачу позволяют такие команды

```
Add             R1,R3
AddWithCarry    R2,R4
```

Между указанными командами имеется неявная зависимость, поскольку обе используют флаг переноса. Флаг устанавливается первой командой и анализируется второй командой, выполняющей операцию

$$R4 \leftarrow [R2] + [R4] + \text{перенос}$$

Когда применяются команды с побочными эффектами, количество зависимостей между данными может стать таким, что для разрешения конфликта потребуется существенно усложнить аппаратное и программное обеспечение. Поэтому для конвейерных процессоров разрабатываются команды, у которых число побочных эффектов сведено минимуму. В идеале команда должна изменять только результирующий операнд в регистре или в основной памяти. Желательно, чтобы побочные эффекты -- установка флагов условий и обновление содержимого указателя — были минимальными. Однако, как указывалось в главе 2, автоинкрементный и автодекрементный режимы адресации очень полезны. Флаги условий тоже нужны, поскольку в них записывается информация о важных событиях: переносе или переполнении в результате арифметических операций. В разделе 8.4 рассказывается, как реализовать эти функции, чтобы они были совместимы с конвейерной организацией процессора и отвечали требованиям оптимизирующих компиляторов.

8.3. Конфликты по управлению

Задача блока выборки команды состоит в обеспечении остальных блоков непрерывным потоком команд. Если этот поток прерывается, конвейер останавливается, как в случае промаха при обращении к кэш-памяти (рис. 8.4). Приостановить работу конвейера может и команда перехода. В следующем разделе обсуждается влияние команд перехода на функционирование конвейера и рассматриваются пути его уменьшения.

8.3.1. Безусловный переход

На рис. 8.8 показана последовательность команд, выполняемая на двухступенчатом конвейере. Команды с I_1 по I_3 хранятся по последовательным адресам памяти. Назначение команды I_2 — безусловный переход. Предположим, что переход выполняется к команде I_k. На такте 3 одновременно с декодированием команды перехода производится выборка команды I_3. На такте 4 процессор должен удалить команду I_3, в выборе которой не было необходимости, и извлечь команду I_k. Тем временем аппаратный блок, отвечающий за шаг выполнения (E), получает команду приостановить свою работу до конца данного такта. Таким образом, работа конвейера приостанавливается на один такт.

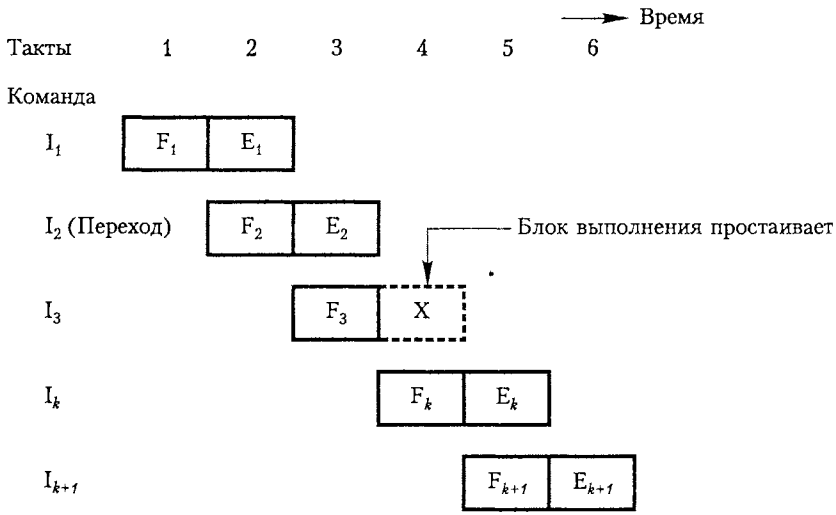


Рис. 8.8. Такт простоя, вызванного командой перехода

Временные потери, понесенные из-за команды перехода, называют *накладными расходами перехода*. На рис. 8.8 накладные расходы составляют один такт. Для более длинного конвейера они могут быть и большими. В качестве примера на рис. 8.9, *а* показано, как выполнение команды перехода отражается на работе 4-ступенчатого конвейера. Предполагается, что адрес перехода вычисляется на шаге E_2. Затем требуется удаление команд I_3 и I_4, а также выбор команды I_k на такте 5. Итак, накладные расходы перехода в этом примере составляют два такта.

Для сокращения накладных расходов перехода адрес перехода нужно вычислять раньше. Обычно в блоке выборки команды имеется выделенная подсистема, предназначенная для быстрого выявления команд перехода и вычисления их целевых адресов. Благодаря этой подсистеме обе задачи можно решить на шаге D_2, результатом чего будет последовательность событий, показанная на рис. 8.9, *б*. В этом случае накладные расходы перехода составят один такт.

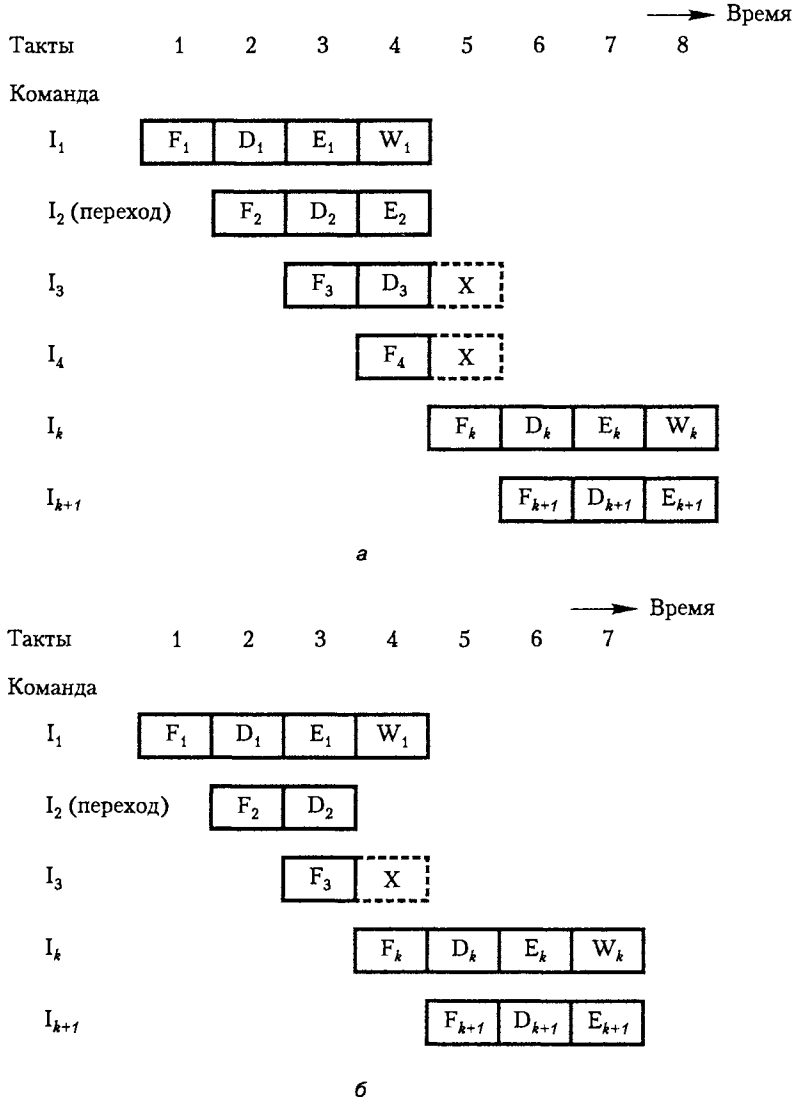


Рис. 8.9. Временные диаграммы работы конвейера при выполнении команды условного перехода: адрес перехода вычисляется на ступени выполнения (*а*); адрес перехода вычисляется на ступени декодирования (*б*)

Очередь команд и упреждающая выборка

Промахи при обращении к кэш-памяти и команды перехода обуславливают приостановку конвейера на один и более тактов. Для сокращения отрицательных последствий этих событий во многие процессоры включают сложные блоки выборки, которые извлекают команды еще до того, как они понадобятся, и помещают их в очередь. Отдельный блок, называемый *блоком диспетчеризации*, пересылает команды, расположенные в начале очереди, блоку выполнения. Схема обработки команд таким процессором приведена на рис. 8.10. Помимо выборки команд из очереди блок диспетчеризации выполняет их декодирование.

Чтобы конвейер функционировал эффективно, блок выборки должен обладать мощными средствами декодирования и обработки команд, позволяющими распознавать и выполнять команды перехода. Его задача — постоянное формирование очереди команд, что уменьшит влияние на работу конвейера случайных задержек при выборке очередных команд. Если останов конвейера вызван конфликтом по данным, блок диспетчеризации не может передавать следующим ступеням команды из очереди. Это, однако, не мешает блоку выборки продолжать извлекать команды и помещать их в очередь. Когда задержка при выборке команды возникает из-за перехода или промаха во время обращения к кэшу, блок диспетчеризации может продолжать извлекать команды из очереди и передавать их следующим блокам для выполнения.

На рис. 8.11 показано, как изменяется длина очереди и каким образом это отражается на взаимодействии ступеней конвейера. Предполагается, что изначально очередь содержит одну команду. При выполнении операции выборки очередь увеличивается на одну команду, а при выполнении операции диспетчеризации — уменьшается также на одну команду. Поэтому в течение первых четырех тактов длина очереди остается неизменной. (На каждом из этих тактов выполняются шаги F и D.) Допустим, команда I_1 вызывает останов конвейера на два такта. Поскольку в очереди еще есть место, блок выборки продолжает свою работу, а длина очереди на такте 6 увеличивается до 3.

Команда I_5 является командой перехода. Ее целевая команда I_k выбирается на такте 7, а команда I_6 удаляется из конвейера на том же такте. Но на этот раз удаление команды I_6 не приводит к останову конвейера на такте 7, как при отсутствии очереди команд. Вместо этого команда I_4 диспетчеризируется из очереди на ступень дешифрирования. После удаления команды I_6 длина очереди на такте 8 сокращается до одной команды и остается такой, пока не произойдет следующий останов конвейера.

Теперь изучим процесс выполнения команд (рис. 8.11). Команды I_1, I_2, I_3, I_4 и I_k прекращают действовать на последовательных тактах. При этом команда перехода не увеличивает общее время выполнения, поскольку выбор целевой команды I_k осуществляется одновременно с выполнением других команд. С этой целью ее адрес вычисляется еще при выборке команды перехода. Описанная технология обработки переходов называется *ветвлением с совмещением*.

Обратите внимание на то, что ветвление с совмещением может быть произведено только в том случае, если к моменту обнаружения процессором команды перехода в очереди будет находиться еще хотя бы одна команда. При отсутствии других команд выполнение продолжится так, как показано на рис. 8.9, *б*. Поэтому

следует поддерживать очередь заполненной, чтобы из нее в любой момент можно было выбрать команду для обработки. Для этого можно повысить скорость извлечения команд из кэша блоком выборки. Во многих процессорах ширина шины между блоком выборки команд и кэшем команд позволяет считывать более одной команды на одном такте. Если после перехода блок выборки снова быстро заполнит очередь команд, вероятность ветвления с совмещением увеличится.

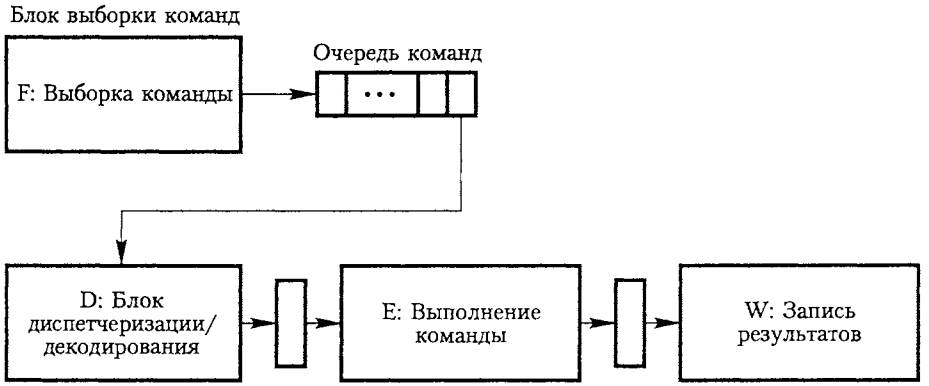


Рис. 8.10. Очередь команд для конвейера, показанного на рис. 8.2, *б*

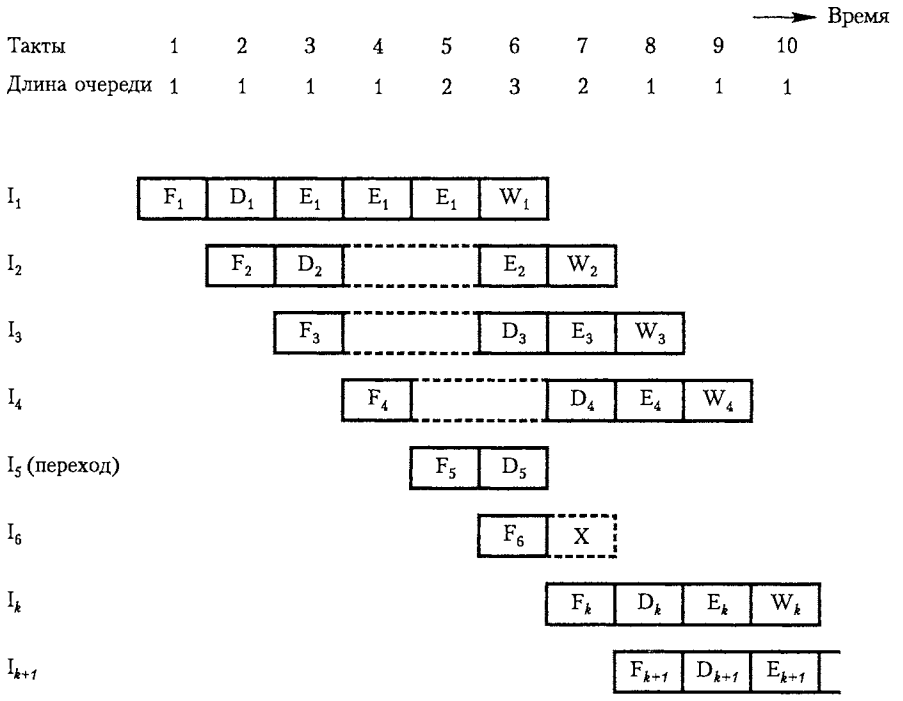


Рис. 8.11. Выполнение перехода при наличии очереди команд; целевой адрес перехода вычисляется на ступени D

Наличие очереди команд положительно сказывается и на обработке промахов при обращении к кэш-памяти. Когда происходит промах, блок диспетчеризации продолжает направлять команды для выполнения, пока очередь не опустеет. Тем временем нужная информация считывается из основной памяти или кэш-памяти второго уровня. При возобновлении операций выборки очередь команд заполняется снова. Если очередь не опустеет до получения данных, промах, возникший при обращении к кэшу, не отразится на скорости выполнения команд.

Таким образом, очередь команд и технология ветвления с совмещением позволяют снизить отрицательное влияние команд перехода и промахов на производительность компьютера при обращении к кэш-памяти. Эффективность этой технологии тем выше, чем больше команд может быть одновременно считано из кэш-памяти.

8.3.2. Условные переходы и предсказание переходов

Команды условного перехода могут послужить причиной дополнительных конфликтов, связанных с тем, что условие перехода зависит от результата выполнения предшествующей команды. Пока ее обработка не завершится, процессор не примет решение о переходе.

Команды перехода часто встречаются в программах. В типичной программе они составляют около 20 % общего (динамического) количества команд (включая и те, которые выполняются многократно). «Накладные расходы», связанные с командами перехода, при таком их количестве приводят к существенному снижению производительности компьютера (по сравнению с той, которую можно было бы достичь благодаря конвейерной обработке команд при отсутствии команд перехода). Отрицательное влияние команд перехода на производительность можно существенно уменьшить за счет применения специальных методов их обработки.

Отложенный переход

Согласно рис. 8.8, процессор выбирает команду I_3 до того, как определит, является ли текущая команда I_2 командой перехода. Когда выполнение команды I_2 завершится и настанет время перехода, процессор должен будет удалить команду I_3 из конвейера и выбрать в памяти целевую команду перехода. Команда, следующая за командой перехода, называется *слотом задержки перехода*. Количество слотов зависит от того, сколько времени требуется для выполнения команды перехода. Например, на рис. 8.9, *а* смоделирована ситуация, когда применяется два слота задержки, а на рис. 8.9, *б* — один. В процессе вычисления адреса перехода команды, помещаемые в слоты задержки, могут извлекаться из памяти и даже частично выполняться.

Технология *отложенного перехода* позволяет минимизировать накладные расходы команд условного перехода. Концепция, на которой она основана, проста. Команды в слотах задержки *всегда* выбираются из памяти, поэтому можно рассчитывать, что они будут выполнены независимо от того, произойдет переход или нет. Эти слоты желательно заполнить полезными командами, а если таковых не

найдется, то командами NOP. Описанная ситуация напоминает ту, которую мы обсуждали в разделе 8.2, когда говорили о зависимости данных.

Рассмотрим последовательность команд, представленную на рис. 8.12, *а*. Регистр R2 применяется в качестве счетчика количества операций сдвига влево содержимого регистра R1. Для процессора с одним слотом задержки эти команды можно реорганизовать так, как показано на рис. 8.12, *б*. Команда сдвига выбирается во время выполнения команды перехода. После вычисления условия перехода процессор выбирает команду по адресу LOOP или NEXT, в зависимости от того, истинно или ложно условие перехода. В любом случае он завершает выполнение команды сдвига. Последовательность событий во время двух последних проходов по циклу указана на рис. 8.13. Конвейер работает непрерывно. Логически программа выполняется так, как будто команда перехода следует за командой сдвига. Это означает, что переход происходит на одну команду позднее, чем при последовательном выполнении команд программы. Отсюда и название — *отложенный переход*.

LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

а

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift_left	R1
NEXT	Add	R1,R3

б

Рис. 8.12. Изменение последовательности команд для отложенного перехода: исходный цикл программы (*а*); переупорядоченные команды (*б*)

Эффективность технологии отложенного перехода зависит от того, насколько часто удается переупорядочить команды (рис. 8.12). Анализ многих программ показывает, что при использовании сложных технологий компиляции один слот задержки можно заполнить в 85 % случаев. Если процессор имеет два слота задержки, компилятор старается найти перед командой перехода две команды, которые можно поместить в эти слоты, не вызвав логической ошибки. Вероятность того, что такие команды найдутся, невелика. Так что, если вместе с количеством ступеней конвейера увеличивается число слотов задержки, потенциальное повышение производительности может быть не достигнуто.

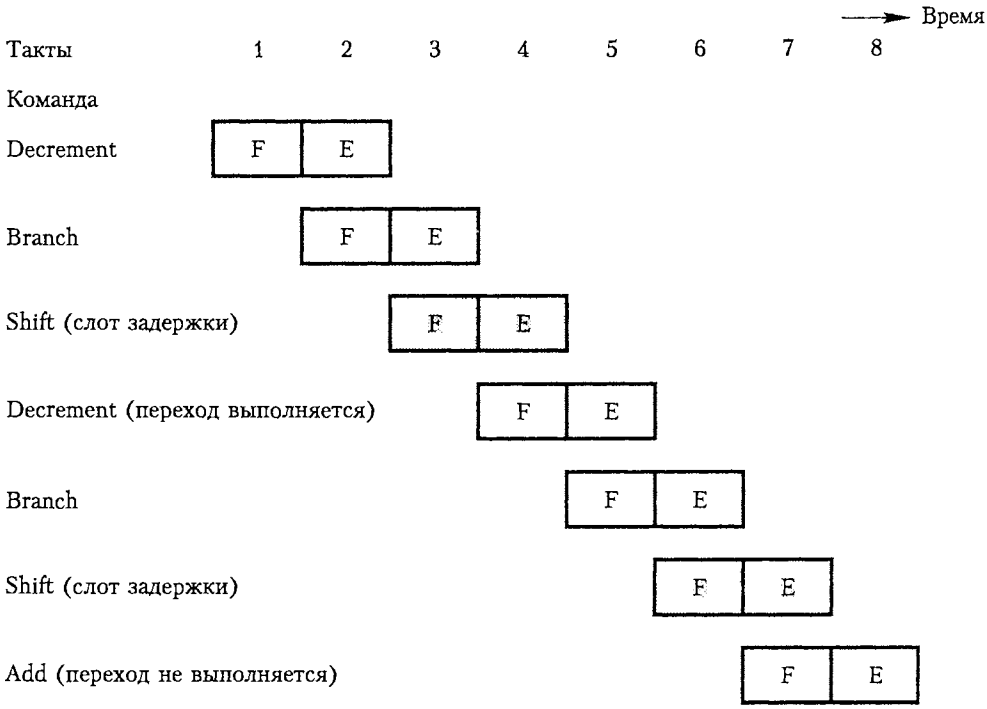


Рис. 8.13. Временная диаграмма, демонстрирующая заполнение слота задержки на двух последних итерациях цикла (рис. 8.12, *б*)

Предсказание переходов

Еще одна технология сокращения накладных расходов условных переходов основана на предсказании выполнения или невыполнения конкретного перехода. Рассмотрим простейшую форму. В этом случае предполагается, что перехода не будет, и продолжается выбор команд программы с последовательными адресами. Пока условие перехода не проверено, происходит *упреждающее выполнение* выбранных команд: команды обрабатываются до того, как процессор «удостоверится», что они выбраны в правильном порядке. Однако ни память, ни регистры процессора не обновляются до получения подтверждения, что эти команды и в самом деле должны быть выполнены. Если проверка условия перехода показывает обратное, команды и все связанные с ними данные удаляются из блоков выполнения, после чего выбираются и приводятся в действие нужные команды программы.

Пример неправильно предсказанного перехода для 4-ступенчатого конвейера приведен на рис. 8.14. Здесь показано, что выполняется команда сравнения, за которой следует команда Branch>0. Прогнозирование направления перехода выполняется на такте 3 одновременно с выборкой команды I_3. Блок выборки предсказывает, что переход не произойдет, и когда команда I_3 переходит на ступень декодирования, он выбирает команду I_4. Результаты операции сравнения становятся доступными по завершении такта 3. Если они перенаправляются в блок выборки команд немедленно, условие перехода проверяется на такте 4. Блок выборки

команды обнаруживает, что предсказание перехода было неверным, и две активные команды удаляются из конвейера. На такте 5 из памяти по целевому адресу перехода выбирается команда I_k.

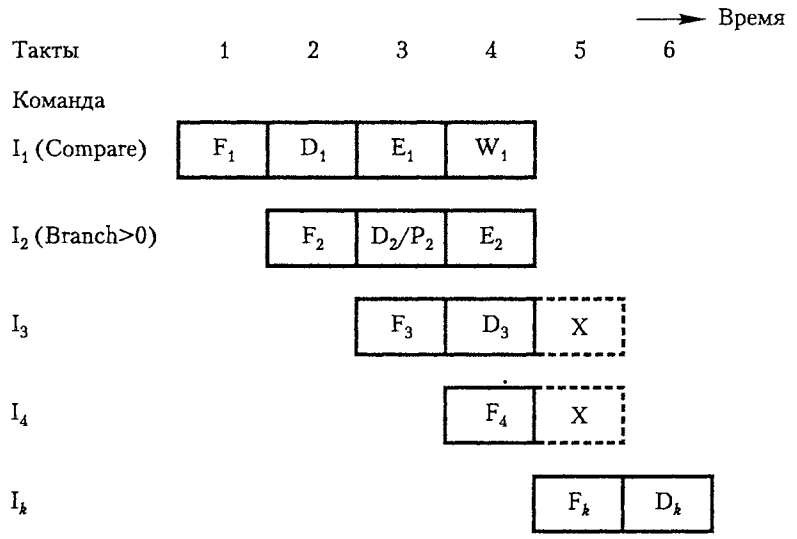


Рис. 8.14. Временная диаграмма неправильно предсказанного перехода

Если условия перехода выполняются и не выполняются с равной вероятностью, то в половине случаев перехож не осуществляется и экономится 50 % времени, которое теряется при условных переходах. Давайте исходить из того, что предсказание направления перехода основывается на ожидаемом поведении программы, и в одних случаях предполагается, что переход произойдет, а в других, что его не будет. В результате, если прогноз будет правильным более чем в половине случаев, можно достичь и более высокой производительности. Допустим, команда перехода, находящаяся в конце цикла и задающая переход к его началу, выполняет переход на каждой итерации цикла, кроме последней. Есть основания предполагать, что переход будет выполнен, значит, можно выбирать команды, следующие за целевым адресом перехода. А вот если команда перехода расположена в начале цикла, перехода, скорее всего, не будет.

Решение о направлении перехода может приниматься аппаратным обеспечением, что зависит от того, располагается целевой адрес перехода выше или ниже адреса команды перехода. Более гибкий подход заключается в том, чтобы возложить эту задачу на компилятор, который имеет больше сведений о программе. Команды перехода некоторых процессоров, например SPARC, содержат разряд предсказания перехода, устанавливаемый компилятором в 0 или 1. Блок выборки команды проверяет этот разряд и выполняет действия в соответствии с указанным в нем предсказанием.

В случае использования любой из этих схем каждый раз, когда выполняется команда условного перехода, прогнозируемое направление перехода не изменяется.

Такой метод называется *статическим предсказанием перехода*. Другой метод, *динамическое предсказание перехода*, заключается в том, что прогнозируемое направление перехода изменяется в зависимости от истории выполнения программы.

Динамическое предсказание перехода

Задачей любого алгоритма предсказания перехода является снижение вероятности принятия неверного решения с целью избежать выборки команд, которые через некоторое время будут отменены и удалены из конвейера. В случае динамического предсказания перехода аппаратное обеспечение процессора оценивает, насколько велика его вероятность, основываясь на результатах выполнения команды перехода. В простейших алгоритмах история охватывает только результат последнего выполнения команды. Процессор предполагает, что при следующей активизации команды результаты будут такими же. Опишем алгоритм, о котором идет речь, на примере конечного автомата с двумя состояниями (рис. 8.15, *а*). Характеристика его состояний дана ниже.

БП: Большая вероятность перехода

МП: Малая вероятность перехода

Предположим, что изначальным является состояние МП. Если в результате активизации команды перехода он действительно происходит, автомат переходит в состояние БП, иначе сохраняется состояние МП. Когда такая же команда встретится еще раз, будет предсказано выполнение перехода, если автомат находится в состоянии БП, и его невыполнение в противном случае.

Представленная простая схема, в которой каждой команде перехода нужен всего один разряд информации об истории ее выполнения, является удачным решением для программных циклов. После входа в цикл управляющие им команды перехода всегда дают один и тот же результат вплоть до выхода из цикла. На последней итерации предсказание перехода окажется неправильным, вследствие чего произойдет смена состояний автомата. К сожалению, это означает, что при следующем входе в тот же цикл в случае выполнения хотя бы одного прохода первое предсказание окажется неверным.

Производительность становится еще выше при сохранении большего объема информации об истории выполнения команд перехода. На рис. 8.15 представлен алгоритм, рассчитанный на четыре состояния, для которых в командах перехода должно выделяться по два разряда. Ознакомимся с описаниями состояний.

ОБП: Очень большая вероятность перехода

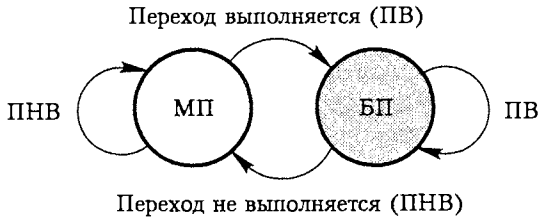
БП: Большая вероятность перехода

МП: Малая вероятность перехода

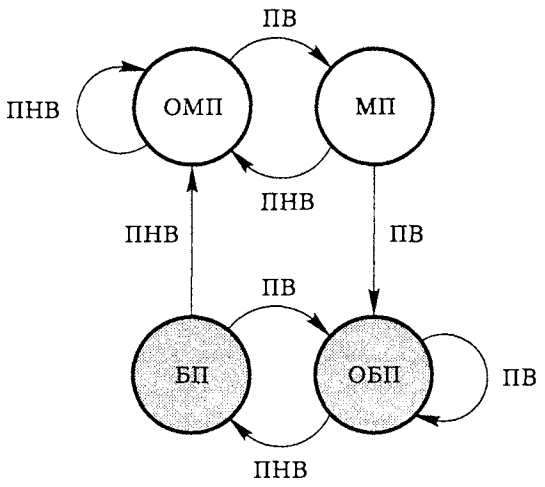
ОМП: Очень малая вероятность перехода

И в этот раз мы будем исходить из того, что изначальным является состояние МП. Когда команда перехода выполнена, автомат перейдет в состояние ОБП, если переход состоится, и в состояние ОМП в противном случае. Когда в ходе работы программы снова встретится та же команда перехода, блок выборки команд предскажет выполнение перехода и начнет выборку команд по целевому адресу,

если автомат находится в состоянии БП или ОБП. Иначе выборка команд будет продолжена в порядке следования адресов.



а



б

Рис. 8.15. Графы переходов конечного автомата с несколькими состояниями для предсказания ветвления: графы переходов конечного автомата с двумя состояниями (*а*); графы переходов конечного автомата с четырьмя состояниями (*б*)

В состоянии ОМП блок выборки команд предсказывает невыполнение перехода. Если же в действительности переход будет выполнен, то есть предсказание окажется неверным, автомат перейдет в состояние МП. Это означает, что, когда данная команда перехода встретится в очередной раз, блок выборки команды снова предскажет невыполнение перехода. И только если предсказание окажется неверным дважды, состояние изменится на ОБП. После этого будет предсказываться выполнение перехода.

Теперь посмотрим, что произойдет при выполнении программного цикла. Предположим, что команда перехода находится в конце цикла и начальным является состояние МП. На первой итерации предсказание (невыполнение перехода)

окажется неверным, и состояние изменится на ОБП. На всех последующих итерациях, кроме последней, предсказание будет верным. Во время последней итерации оно не сбудется, вследствие чего осуществится переход в состояние БП. При следующем входе в тот же цикл предсказание (выполнение перехода) окажется верным.

Чтобы исправить неверное предсказание перехода при первом входе в цикл, достаточно внести в алгоритм одно небольшое изменение. Причиной неверного предсказания является начальное состояние автомата. Не имея дополнительной информации о природе команды перехода, процессор изначально переходит в состояние МП. Для выбора правильного исходного состояния можно использовать информацию, предоставляемую любой из описанных выше схем статического предсказания, то есть либо сравнить адреса, либо проверить разряд предсказания в команде перехода. В обоих случаях начальным состоянием автомата будет состояние МП либо БП. При переходе в конце цикла компилятор может указать, что следует предсказать выполнение перехода. В этом случае будет задано начальное состояние БП. С такой модификацией предсказание перехода всегда будет верным, за исключением последней итерации цикла. Последнего неверного предсказания не избежать.

Существуют различные способы хранения процессором информации о состоянии, используемой алгоритмами динамического предсказания перехода. В частности, она может быть записана в таблицу, для доступа к которой применяются младшие разряды адреса команды перехода. В этом случае есть вероятность применения одной и той же записи таблицы двумя командами перехода, что чревато неверным предсказанием перехода, но не приведет к ошибке выполнения программы. Неправильное предсказание перехода вызывает лишь задержку в выполнении программы. Еще один способ основан на хранении данных о предыдущих переходах в теге, связанном с командой перехода, в кэше команд. В разделе 8.7 вы узнаете, как эта информация обрабатывается в процессоре SPARC.

8.4. Конвейерная обработка и система команд

Не все команды одинаково хорошо подходят для конвейерного выполнения. Например, использование команд с побочными эффектами сопряжено с нежелательными зависимостями между данными. В настоящем разделе мы поговорим о связи между конвейерной обработкой команд и двумя важными характеристиками системы команд компьютера — режимами адресации и поддержкой флагов условий.

8.4.1. Режимы адресации

Набор поддерживаемых компьютером режимов адресации должен обеспечивать простой и эффективный доступ к самым разнообразным структурам данных. Широкую популярность завоевали индексный, косвенный, автоинкрементный и автодекрементный режимы. Многие процессоры позволяют по-разному их комбинировать, благодаря чему их системы команд являются более гибкими.

При выборе режимов адресации для конвейерного процессора следует принимать во внимание их воздействие на поток команд в конвейере. Особенно важно учитывать побочные эффекты автоинкрементной и автодекрементной адресации и вероятность приостановок конвейера из-за использования сложных режимов адресации. Кроме того, имеет значение, насколько часто тот или иной режим адресации будет применяться компилятором.

Для сравнения различных подходов к выбору режимов адресации давайте рассмотрим простую модель доступа к операндам в памяти. Выполнение команды Load X(R1),R2, предназначенной для загрузки данных из памяти, занимает пять тактов (рис. 8.5). Подобную команду

```
Load  (R1),R2
```

способен выполнить 4-ступенчатый конвейер, поскольку она не требует вычисления адреса. Доступ к памяти может осуществляться на ступени E. В случае использования более сложного режима адресации при доступе к операнду не исключена необходимость в нескольких обращениях к памяти. Например, команда

```
Load  (X(R1)),R2
```

может быть выполнена так, как показано на рис. 8.16, *а*, если считать, что смещение индекса, X, задано в слове команды. После вычисления адреса на такте 3 процессору нужно дважды обратиться к памяти — прочитать слово по адресу X+[R1] на такте 4, а также слово по адресу [X+[R1]] на такте 5. Если содержимое регистра R2 является исходным операндом следующей команды, выполнение команды будет задержано на три такта. Эту цифру можно сократить до двух за счет продвижения операнда.

Для выполнения такой же операции загрузки с использованием только простейших адресных режимов потребуется несколько команд. На компьютере, где допускается использование трех операндов, это могут быть такие команды:

```
Add   #X,R1,R2
Load  (R2),R2
Load  (R2),R2
```

Команда сложения выполняет операцию R2 ← X + [R1], а команды загрузки выбирают из памяти сначала адрес, а затем операнд. Этой последовательности команд требуется то же количество тактов, что и первой команде загрузки (рис. 8.16, *б*).

Рассмотренный пример демонстрирует, что использование в конвейерном процессоре сложных режимов адресации, нуждающихся в нескольких обращениях к памяти, не всегда способствует ускорению работы программы. Их главными достоинствами являются сокращение количества команд, необходимых для решения конкретной задачи, и уменьшение занимаемого программой объема основной памяти. Наряду с указанными положительными характеристиками команды, реализующие сложные режимы адресации, имеют и недостаток, состоящий в их длительном выполнении, что приводит к задержке в работе конвейера и снижает ее эффективность. Кроме того, для их декодирования и выполнения необходимо аппаратное обеспечение соответствующей мощности. Да и компиляторам непросто работать с такими командами.

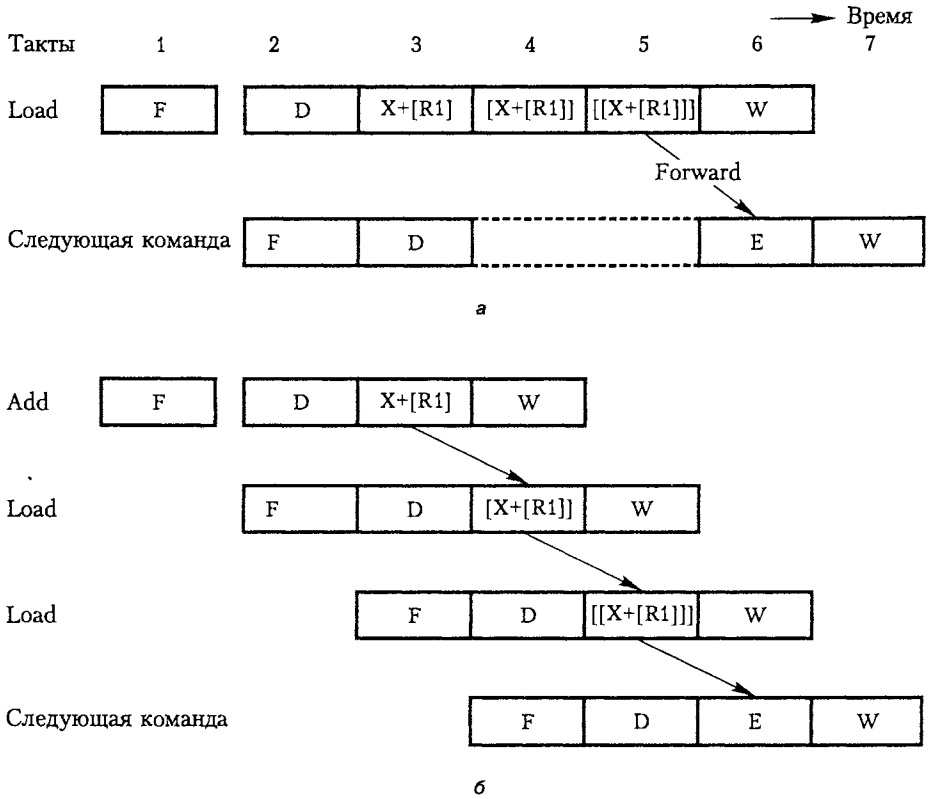


Рис. 8.16. Эквивалентные операции с использованием режимов адресации: сложного (*а*); простого (*б*)

Системы команд современных процессоров ориентированы на максимальное использование преимуществ конвейерной организации процессора. И поскольку сложные режимы адресации не годятся для конвейерного выполнения, их стараются не применять. Адресные режимы современных процессоров соответствуют следующим требованиям:

- Для доступа к операндам выполняется не более одного обращения к памяти.
- Обращение к памяти осуществляется только в командах загрузки данных из памяти (Load) и сохранения данных в памяти (Store).
- Используемые адресные режимы не имеют побочных эффектов.

Этим требованиям соответствуют регистровый, косвенный регистровый и индексный режимы адресации. Первые два вообще не требуют вычисления адреса. В индексном режиме адрес может быть вычислен за один такт независимо от того, задано значение индекса в команде или регистре. Обращение к памяти производится на следующем такте. Перечисленные режимы адресации не имеют побочных эффектов за одним исключением. В некоторых архитектурах, в частности в процессорах ARM, разрешается записывать вычисленный в индексном

режиме адрес обратно в индексный регистр. Это побочный эффект, нарушающий приведенные выше требования. Кроме того, обратите внимание на возможность использования относительной адресации, являющейся одной из разновидностей индексной адресации, при которой в качестве индексного регистра используется счетчик команд.

Три указанных выше правила впервые были реализованы в RISC-процессорах. В разделе 8.7 вы познакомитесь с архитектурой SPARC, которая удовлетворяет выдвинутым в них требованиям.

8.4.2. Коды условий

Во многих процессорах, включая описанные в главе 3, флаги условий хранятся в регистре состояния процессора. Они устанавливаются и очищаются многими командами. В результате их можно проверить в последующих командах условного перехода для изменения потока выполнения программы. Оптимизирующий компилятор конвейерного процессора старается реорганизовать команды программы таким образом, чтобы по возможности избегать остановов конвейера из-за переходов или зависимостей между данными последовательных команд. Компилятор должен гарантировать, что реорганизация команд не приведет к изменению результатов вычислений. Зависимости, связанные с использованием флагов условий, мешают компилятору реорганизовывать команды.

В качестве примера рассмотрим последовательность команд, приведенную на рис. 8.17, *а*. Предположим, что команды Compare и Branch=0 выполняются так, как показано на рис. 8.14. Решение о переходе принимается не на шаге D_2, а на шаге E_2, поскольку для него нужны результаты обработки команды Compare. Время, необходимое команде перехода, можно сократить, поменяв местами команды Add и Compare (рис. 8.17, *б*). В результате во время декодирования команды Branch уже будут доступны результаты выполнения команды сравнения, что позволит сразу принять решение о переходе без предсказаний. Важный момент заключается в том, что команды сложения и сравнения можно поменять местами только в том случае, если команда сложения не воздействует на флаги условий.

```
Add          R1,R2
Compare      R3,R4
Branch=0     ...
```

а

```
Compare      R3,R4
Add          R1,R2
Branch=0     ...
```

б

Рис. 8.17. Изменение порядка следования команд: фрагмент программы (*а*); команды после переупорядочения (*б*)

Проанализировав пример, можно сделать два важных вывода относительно обработки флагов условий. Во-первых, для обеспечения гибкой реорганизации команд флаги условий должны изменяться как можно меньшим количеством команд. Во-вторых, нужно, чтобы компилятор указывал, в каких командах флаги условий модифицируются, а в каких нет. Системы команд, разрабатываемые с учетом конвейеризации, обычно отвечают этим требованиям. Реорганизация команд, представленная на рис. 8.17, *б*, может быть выполнена лишь при условии, что флаги обрабатываются только в том случае, когда это явно указано в коде операции. Такую возможность поддерживают архитектуры SPARC и ARM.

8.5. Тракты данных и управление

С общими принципами организации тракта данных процессора вы познакомились в главе 7. Давайте вернемся к трехшинной структуре, показанной на рис. 7.8. Чтобы приспособить эту структуру к конвейерному выполнению команд, нужно модифицировать ее так, как показано на рис. 8.18. На рисунке вы видите структуру для 4-ступенчатого конвейера. Обращения к кэшу данных могут выполняться на ступени E или на более поздних ступенях, что зависит от режима адресации и особенностей процессора. Рассмотрим важнейшие изменения, внесенные в схему.

1. Отдельные кэши команд и данных, которые имеют отдельные соединения с процессором для передачи адреса и данных. В этом случае используются две версии регистра MAR: IMAR обеспечивает доступ к кэшу команд, а DMAR — к кэшу данных.
2. Регистр PC соединен непосредственно с регистром IMAR, так что его содержимое может быть переслано в IMAR одновременно с выполнением другой операции в АЛУ.
3. Для использования косвенной регистровой и индексной адресации адрес данных, хранящийся в регистре DMAR, можно получить прямо из регистрового файла или из АЛУ.
4. Операциям чтения и записи выделены раздельные регистры MDR. В ходе загрузки и сохранения возможен непосредственный обмен данными между регистрами и регистровым файлом без пропуска через АЛУ.
5. На входах и выходах АЛУ имеются буферные регистры (на рис. 8.7 — SRC1, SRC2 и RSLT). Соединения для продвижения данных на рис. 8.18 не показаны.
6. Регистр команд заменен очередью команд, загружаемых из кэша.
7. Выход дешифратора команды соединен с конвейером управляющих сигналов. О необходимости буферизации управляющих сигналов и пересылки их от одной ступени к другой вместе с командами рассказывалось в разделе 8.1. Этот конвейер сохраняет управляющие сигналы в буферах B2 и B3 (рис. 8.2, *а*).

Процессор (рис. 8.18), может независимо выполнять следующие операции:

- ✦ считывать команды из кэша команд;
- ✦ производить приращение значений регистра PC;

- декодировать команды;
- считывать из кэша и записывать в кэш данные;
- считывать содержимое двух регистров из регистрового файла;
- осуществлять запись в один регистр регистрового файла;
- выполнять операции АЛУ.

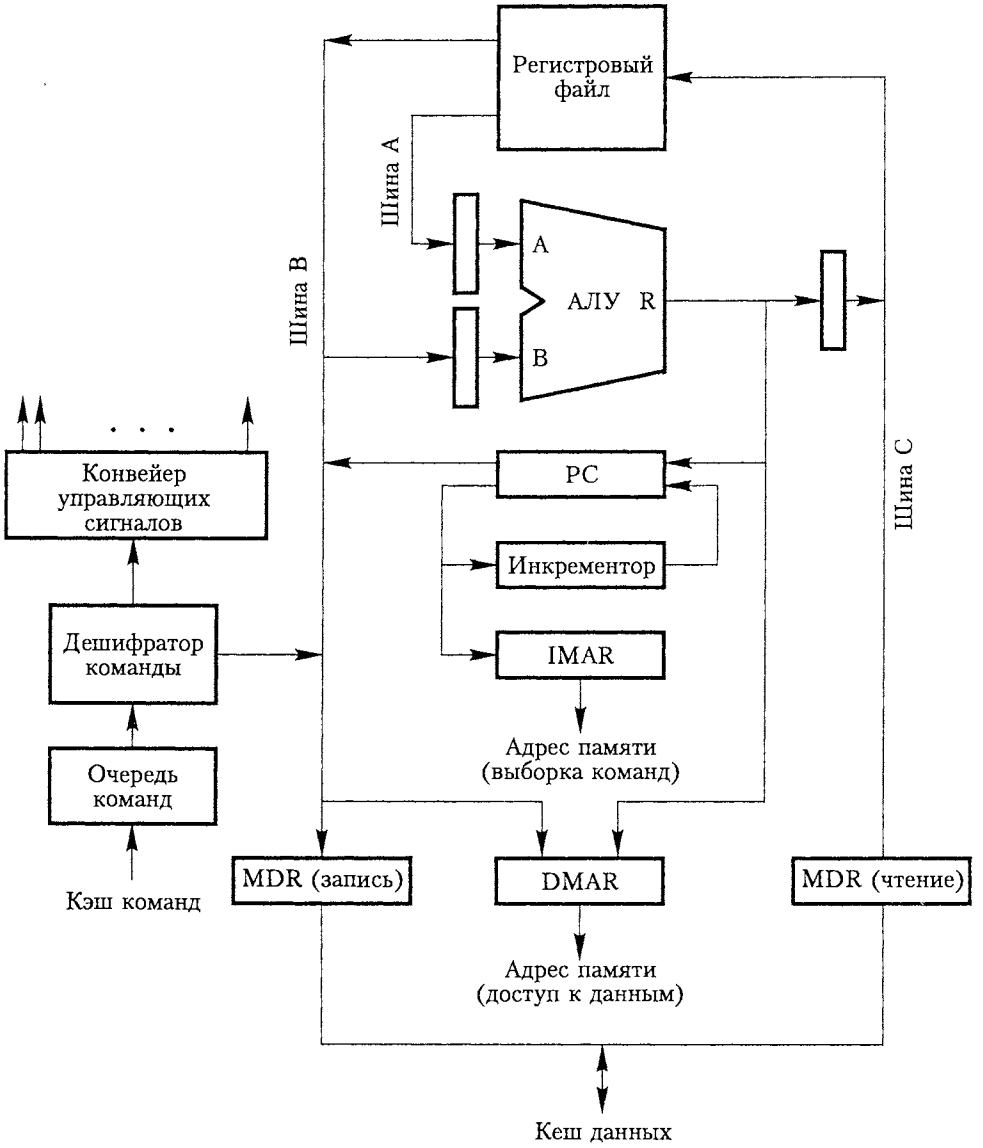


Рис. 8.18. Тракт данных, модифицированный для конвейерного выполнения с промежуточными буферами на входе и выходе АЛУ

Поскольку в перечисленных операциях не задействованы общие ресурсы, их можно выполнять одновременно в любых сочетаниях. Это обеспечивает гибкость, необходимую для реализации 4-ступенчатого конвейера. В качестве примера возьмем последовательность из четырех команд: I_1, I_2, I_3 и I_4. Как показано на рис. 8.2, *а*, на такте 4 производятся следующие действия:

- запись результата команды I_1 в регистровый файл;
- чтение операндов команды I_2 из регистрового файла;
- декодирование команды I_3;
- выборка команды I_4 и приращение значения регистра PC.

8.6. Суперскалярная обработка команд

Конвейеризация обеспечивает параллельную обработку команд. При использовании этой технологии конвейер содержит несколько команд, находящихся на разных ступенях выполнения. Пока первая команда производит операцию АЛУ, вторая декодируется, а третья выбирается из памяти. Команды поступают в конвейер в том порядке, в каком они располагаются в программе. При отсутствии конфликтов на каждом такте в конвейере завершается выполнение очередной команды и появляется новая команда. Таким образом, максимальная пропускная способность конвейера равна одной команде за такт.

С целью повышения производительности процессор может быть оборудован несколькими обрабатывающими устройствами, чтобы на каждом из них обрабатывалось параллельно несколько команд. При такой организации процессора на одном такте может быть запущено на выполнение нескольких команд. Процессоры такого типа называются *суперскалярными*. Суперскалярную архитектуру имеют многие современные высокопроизводительные процессоры.

Как отмечалось в разделе 8.3, чтобы поддерживать очередь команд заполненной, процессор должен иметь возможность выбирать из кэша несколько команд за раз. Это особенно важно для суперскалярного режима выполнения команд, которому требуется более широкое соединение с кэш-памятью и несколько блоков выполнения. В частности, целочисленным командам и командам с плавающей запятой в суперскалярном процессоре отведены раздельные блоки выполнения.

На рис. 8.19 приведен пример процессора с двумя блоками выполнения: для целочисленных операций и операций с плавающей запятой. Блок выборки команд способен считывать из кэша по две команды за раз и сохранять их в очереди. На каждом такте блок диспетчеризации извлекает из очереди и декодирует одну или две команды. Если одна из команд обрабатывает целочисленные значения, а другая — числа с плавающей запятой, при отсутствии конфликтов обе команды диспетчеризируются на одном такте.

В суперскалярном процессоре конфликты сильнее влияют на производительность, чем в обычном конвейерном процессоре. Компилятор предотвращает многие конфликты, оптимальным образом выбирая и переупорядочивая команды.

Например, для процессора, показанного на рис. 8.19, он может обеспечить чередование операций с плавающей запятой и целочисленных операций. Это позволит блоку диспетчеризации поддерживать непрерывную работу целочисленного арифметического устройства и арифметического устройства с плавающей запятой. В общем случае предельное повышение производительности достигается за счет такого переупорядочения команд компилятором, при котором максимально используются возможности всех доступных устройств.

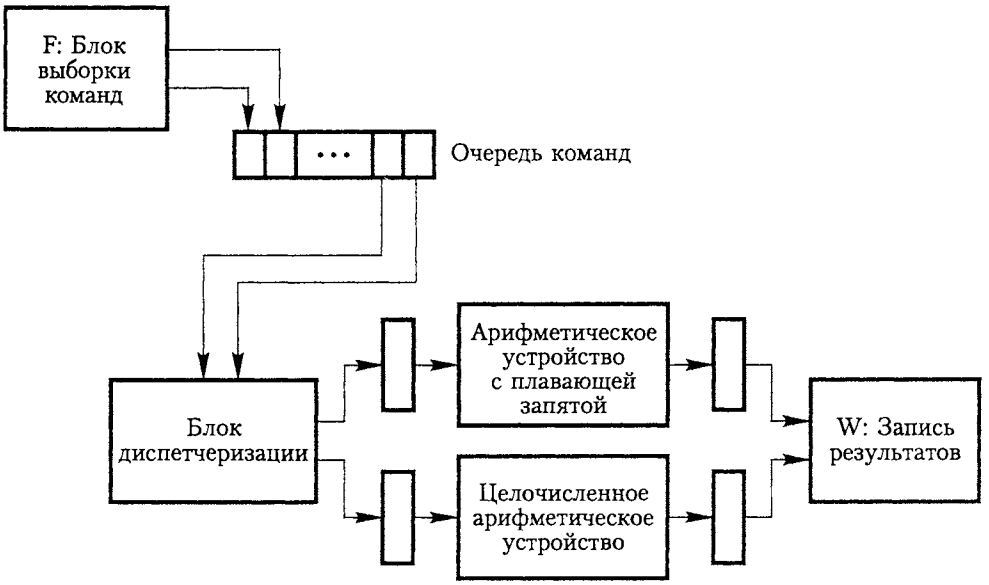


Рис. 8.19. Процессор с двумя блоками выполнения

На рис. 8.20 вы видите временную диаграмму конвейера суперскалярного процессора. Тонкими линиями выделены операции, выполняемые в арифметическом устройстве с плавающей запятой. Для завершения операции, заданной в команде I_1, арифметическому устройству с плавающей запятой требуются три такта. Целочисленное устройство заканчивает выполнение команды I_2 за два такта. Предполагается, что арифметическое устройство с плавающей запятой организовано как трехступенчатый конвейер. Поэтому на каждом такте оно может принимать новую команду. Команды I_3 и I_4 поступают в блок диспетчеризации на такте 3 и диспетчеризируются на такте 4. Целочисленный блок в этот момент ожидает новую команду, так как команда I_2 уже находится на ступени записи. Команда I_1 хотя все еще остается на ступени выполнения, уже перемещена на вторую ступень внутреннего конвейера устройства с плавающей запятой. Поэтому на его первую ступень может поступить команда I_3. Если предположить, что в ходе процесса не произойдет никаких конфликтов, выполнение команд завершится так, как показано на рис. 8.20.

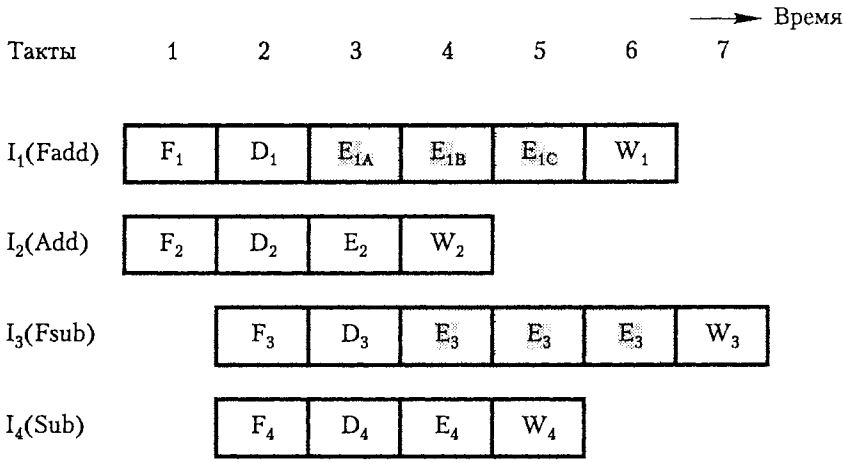


Рис. 8.20. Поток выполнения команд в процессоре с рис. 8.19 при отсутствии конфликтов

8.6.1. Внеочередное завершение команд

Команды, представленные на рис 8.20, диспетчеризируются в том порядке, в каком они расположены в программе. Однако при завершении работы порядок уже иной. Сопряжено ли это с к какими-либо проблемами? Нам с вами приходилось обсуждать вопросы, связанные с зависимостям между командами. Если, к примеру, команда I_2 зависит от результата, выдаваемого командой I_1, выполнение команды I_2 будет отложено. При правильной обработке зависимостей выполнение команды откладывать не приходится. Однако, когда выполнение команды приводит к исключениям, возникают новые проблемы. Исключения могут быть обусловлены сбоем в работе шины в ходе выборки операнда или попыткой совершить недопустимую операцию, например деление на нуль. На такте 4 результаты выполнения команды I_2 записываются в регистровый файл. Если команда I_1 вызывает исключение, команды в конвейере оказываются в несогласованном состоянии. Счетчик указывает на команду, послужившую причиной возникновения исключительной ситуации. При этом выясняется, что одна или несколько следующих за ней команд уже выполнены. Такая ситуация называется *неточным исключением.*

Для того чтобы гарантировать согласованное выполнение программы, результаты выполнения команд нужно записывать по целевым адресам точно в том порядке, в каком они следуют в программе. Это означает, что шаг W_2 на рис. 8.20 необходимо отложить до такта 6. В свою очередь, целочисленное арифметическое устройство должно сохранять результаты выполнения команды I_2, поэтому оно не может принять команду I_4 до шага 6, как показано на рис. 8.21, *а*. Если исключение происходит, когда команда активна, все последующие команды, которые на тот момент, возможно, уже частично выполнены, удаляются из конвейера. Такая обработка исключений называется *точным исключением.*

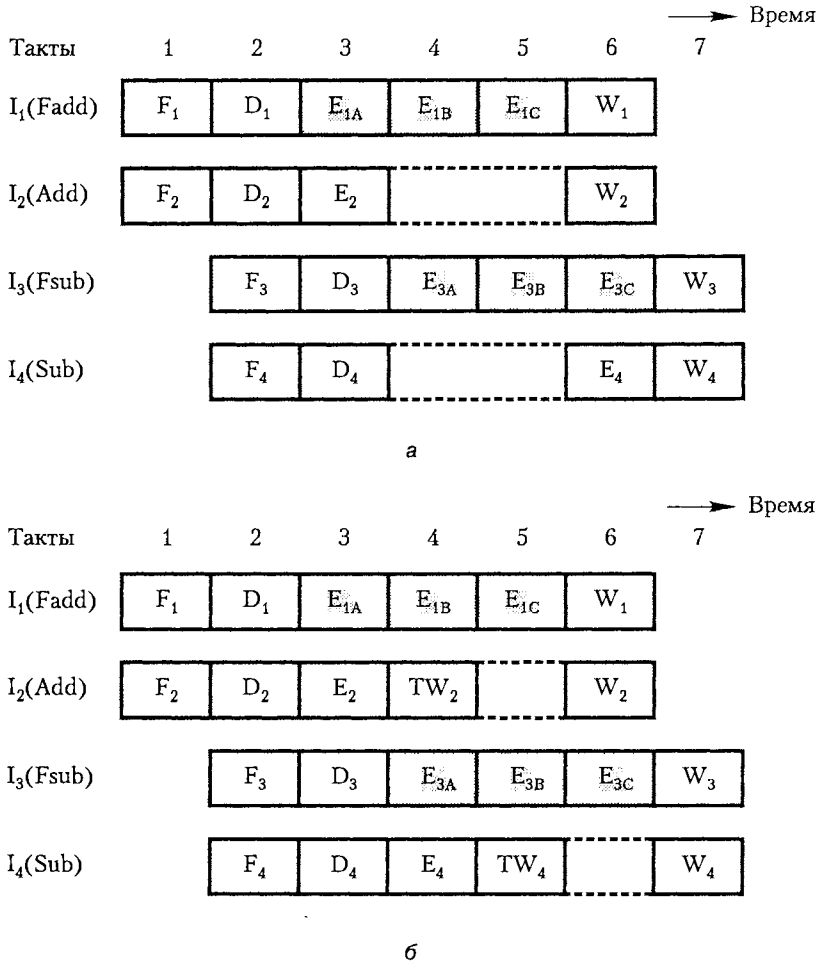


Рис. 8.21. Завершение работы команд в соответствии с порядком их следования в программе: отложенная запись (*а*); использование временных регистров (*б*)

Точные исключения легче обеспечить для внешних прерываний. Как только поступает сигнал внешнего прерывания, блок диспетчеризации прекращает считывать из очереди новые команды и очищает ее. Команды, выполнение которых уже началось, завершают свой рабочий цикл. С этого момента процессор и все его регистры находятся в согласованном состоянии. Обработку исключения можно начинать.

8.6.2. Завершение выполнения

С одной стороны, чтобы ускорить освобождение блоков выполнения и загрузку в них новых команд, можно разрешить внеочередное завершение команд. С другой стороны, для обеспечения точных исключений команды должны завершаться

в том порядке, в каком они следуют в программе. Мы оказались на распутье? Вовсе нет. Существует возможность достичь сразу двух указанных целей. Для этого команды должны выполняться так, как показано на рис. 8.20. Отметим, что результаты при этом записываются во временные регистры. Позднее содержимое временных регистров в нужном порядке пересылается в постоянные регистры процессора. Пример такого варианта выполнения программы приведен на рис. 8.21, *б*. На шаге TW осуществляется запись во временный регистр, а на шаге W, завершающем для команды, содержимое временного регистра переписывается в соответствующий постоянный регистр. Этот шаг часто называют *шагом сохранения*, поскольку после него результат выполнения команды уже не может быть отменен. Если команда вызывает исключение, результаты, выданные последующими командами, находящимися во временных регистрах, легко удалить.

Временный регистр исполняет роль того постоянного регистра, данные которого в нем хранятся. И даже имя ему присваивается такое же. Например, если целевым регистром команды I_2 является R5, временный регистр, используемый на шаге TW_2, на тактах 6 и 7 интерпретируется как R5. Его содержимое передается любым следующим командам, обращающимся к регистру R5. Описанная технология называется *переименованием регистров*. Обратите внимание на то, что временный регистр применяется только теми командами, которые при естественном порядке выполнения программы следуют за I_2. Если на такте 5 или 6 регистр R5 потребуется команде, предшествующей I_2, она получит доступ к реальному регистру R5, который все еще содержит данные, не измененные командой I_2.

В случае допуска внеочередного завершения команд необходимо, чтобы в процессоре присутствовал специальный управляющий блок — *блок сохранения*. Этот блок выбирает следующую команду для сохранения, используя очередь, называемую *буфером реорганизации*. Команды записываются в эту очередь в том порядке, в каком они следуют в программе, по мере диспетчеризации для выполнения. Когда команда достигает начала очереди и завершается, соответствующие результаты пересылаются из временных регистров в постоянные. После этого команда удаляется из очереди, а все выделенные ей ресурсы, включая временные регистры, очищаются. Итак, команда теперь считается *покинувшей конвейер*. Этот статус имеют также все команды, диспетчеризированные до нее. Описанный алгоритм позволяет выполнять команды в любой последовательности и обеспечивает их выход с конвейера в строгом соответствии с порядком расположения в программе.

8.6.3. Операция диспетчеризации

Давайте вернемся к процессу диспетчеризации команд. Принимая решение о диспетчеризации, соответствующий блок должен гарантировать, что все нужные команде ресурсы доступны. Например, должны быть свободны временные регистры, поскольку в них могут записываться результаты выполнения команды. Их резервирование для определенной команды является частью процесса диспетчеризации. Кроме того, необходим доступ к буферу реорганизации. Когда команда получает в свое распоряжение весь спектр ресурсов, включая блок выполнения, она диспетчеризируется для выполнения.

Можно ли диспетчеризировать команды не по порядку? Чтобы ответить на этот вопрос, обратимся к рис. 8.20, б. Если, например, вызов команды I2 будет отложен из-за промаха при обращении к кэш-памяти за ее исходным операндом, на такте 4 целочисленный блок окажется занятым, и подготовка команды I_4 к выполнению станет невозможной. Корректна ли в этом случае диспетчеризация команды I_5? В принципе, да, если для команды I_4 в буфере реорганизации зарезервировано место, что обеспечит выход команд из конвейера в правильном порядке. Однако внеочередная диспетчеризация команд требует большой осторожности. Приступая к ней, нужно иметь гарантию, что не произойдет взаимоблокировки.

Взаимоблокировкой называется ситуация, когда два блока, скажем A и B, совместно используют один ресурс. Предположим, блок B не может завершить работу до тех пор, пока блок A не закончит свою. В то же время блоку B выделен ресурс, который нужен блоку A. При таком раскладе ни один из двух блоков не сможет решить свою задачу. Блок A будет ожидать освобождения ресурса, а блок B, «оккупировавший» нужный ресурс, — завершения работы блока A.

Рассмотрим, вследствие чего происходит взаимоблокировка, если команды диспетчеризируются не по порядку. Допустим, процессор имеет только один временный регистр, который резервируется для команды I_5, когда она диспетчеризируется. Из-за этого нельзя подготовить к выполнению команду I_4, которая ждет освобождения временного регистра, а его невозможно освободить, пока команда I_5 не покинет процессор. В свою очередь, команда I_5 не покинет процессор прежде, чем это сделает команда I_4. Таким образом, налицо взаимоблокировка.

Для предотвращения взаимоблокировок диспетчер должен учитывать множество факторов. Поэтому при обеспечении поддержки внеочередного выполнения команд значительно усложняется блок диспетчеризации. Кроме того, увеличивается время, затрачиваемое на принятие решения о диспетчеризации. Вот почему в большинстве процессоров команды подготавливаются к выполнению по порядку. Порядок соблюдается на выходе из диспетчера и на выходе из конвейера. Между этими двумя точками допускается обработка команд с разной скоростью.

В следующем разделе рассматривается UltraSPARC II — коммерчески успешный суперскалярный процессор с высокой степенью конвейеризации команд. В нем удачно применяются описанные в этой главе технологии и разрешены многие из перечисленных проблем.

8.7. Процессор UltraSPARC II

За последние годы архитектура процессоров претерпела значительные изменения. Классическое разделение процессоров на RISC и CISC кануло в Лету, поскольку современные высокопроизводительные процессоры включают элементы обеих архитектур.

Ранние RISC-процессоры продемонстрировали возможности повышения производительности с помощью нескольких ключевых технологий. Вот наиболее важные из них:

- конвейеризация, позволяющая процессору одновременно выполнять несколько команд, что при условии редких остановов конвейера приводит к значительному повышению производительности;

- тесная интеграция аппаратного обеспечения и компилятора, при которой максимально используются преимущества конвейерной организации компьютера за счет сокращения количества остановов конвейера.

Именно эти две технологии (а не простое сокращение команд) предопределили коммерческий успех процессоров RISC. В этом отношении особенно важна тесная связь между архитектурой аппаратного обеспечения, структурой конвейера и компилятором. Высокую производительность современных компьютеров в значительной степени обеспечивают эффективные технологии компиляции, которые послужили стимулом для широкого использования прогрессивных аппаратных технологий, еще несколько лет назад не находивших себе применения.

Ярким примером современных процессорных технологий является архитектура SPARC, лежащая в основе процессоров рабочих станций Sun. Одна из реализаций этой архитектуры — процессор UltraSPARC II, который мы выбрали предметом обсуждения, потому что он хорошо иллюстрирует и суперскалярную архитектуру, и многие особенности разработки конвейеров. Рассказ об этом процессоре начинается с краткого описания его архитектуры, полный вариант которого вы найдете в документации.

8.7.1. Архитектура SPARC

Аббревиатура SPARC образована от словосочетания Scalalable Processor ARChitecture, что в переводе означает наращиваемая архитектура процессора. Это спецификация системы команд процессора, не зависящая от их аппаратной реализации. Более того, SPARC представляет собой открытую архитектуру, благодаря чему не только Sun Microsystems, но и другие компьютерные компании могут реализовывать ее систему команд.

Впервые спецификация архитектуры Sun была опубликована в 1987 году. В ее основу легли разработки, проводившиеся в начале восьмидесятых годов учеными Калифорнийского университета, Беркли. В этом проекте впервые было введено понятие компьютера с сокращенным набором команд и появилась соответствующая аббревиатура RISC. На основе этой архитектуры совместно с несколькими производителями процессорных микросхем корпорация Sun создала множество процессоров, обладающих различной производительностью. Спецификация архитектуры SPARC контролируется международным консорциумом, который периодически выпускает ее усовершенствованную версию. Последняя версия называется SPARC-V9.

Система команд архитектуры SPARC строго соответствует принципу RISC. В ее спецификации описывается процессор с 64-разрядными адресами памяти. Все команды имеют длину 32 разряда. Поддерживаются целочисленные команды и команды с плавающей запятой.

Существует два регистровых файла: для целочисленных данных и для данных с плавающей запятой. Целочисленные регистры имеют длину 64 разряда. Их количество зависит от реализации и варьируется от 64 до 528. В процессорах SPARC используется схема, называемая *окнами регистров*. В каждый момент прикладная программа видит только 32 регистра с именами от R0 до R31. Первые

восемь — это глобальные регистры, которые доступны всегда. Остальные 24 регистра локальны для текущего контекста.

Регистры с плавающей запятой имеют длину 32 разряда, поскольку, согласно стандарту IEEE (см. главу 6), такова длина чисел с плавающей запятой одинарной точности. В системе предусмотрены команды с плавающей запятой для операций с двойной и четверной точностью. Для хранения операндов двойной точности используются два последовательных регистра с плавающей запятой, а для хранения операндов четверной точности — четыре регистра. В общей сложности процессор содержит 64 регистра, от F0 до F63. Операнды одинарной точности хранятся в регистрах с F0 по F31, а операнды двойной точности — в регистрах F0, F2, F4, ..., F63. Регистры F0, F4, F8, ..., F60 предназначены для операндов четверной точности.

Команды Load и Store

Доступ к памяти имеют только команды загрузки и сохранения, операндами которых являются 8-битовый байт, 16-битовое полуслово и 31-битовое слово. Эти команды способны обрабатывать 64-разрядные операнды двух видов: расширенное слово и двойное слово. Команда LDX (Load extended — загрузка расширенная) загружает из памяти во внутренние регистры процессора 64 бита данных, называемые *расширенным* словом. Двойное слово состоит из двух 32-разрядных слов, которые команда LDD (Load Double — загрузка двойного слова) загружает в два регистра процессора с последовательными номерами. Эти слова помещаются в младшие 32 разряда каждого регистра, старшие разряды заполняются нулями. В команде указывается первый из двух регистров, номер которого должен быть четным. Команды загрузки и сохранения, обрабатывающие двойные слова, полезны при перемещении операндов большой точности между памятью и регистрами с плавающей запятой.

Команды загрузки и сохранения используют один из двух вариантов индексного режима адресации:

1. Исполнительным адресом является сумма содержимого двух регистров:

$$EA = [Radr1] + [Radr2]$$

2. Исполнительным адресом является сумма содержимого одного регистра и заданного в команде операнда:

$$EA = [Radr1] + значение$$

В большинстве случаев такой операнд представляет собой 13-разрядное значение со знаком. Этот знак расширяется до 64 разрядов, а результат добавляется к содержимому регистра Radr1.

Команда загрузки, использующая первый адресный режим, записывается следующим образом:

```
Load  [Radr1+Radr2], Rdst
```

Она генерирует исполнительный адрес [Radr1] + [Radr2] и загружает содержимое памяти, расположенное по этому адресу, в регистр Rdst. Во второй разновидности команды загрузки операнд Radr2 заменяется непосредственным значением:

```
Load  [Radr1+Imm], Rdst
```

Команда сохранения имеет похожий синтаксис. Ее первый операнд определяет исходный регистр, данные которого должны быть сохранены в памяти:

```
Store   Rsrc, [Radr1+Radr2]
Store   Rsrc, [Radr1+Imm]
```

Согласно правилам синтаксиса, рекомендованным для команд SPARC, регистр задается с помощью знака %, за которым следует номер регистра. Обозначения %r2 и %2 соответствуют регистру с номером 2. С целью упростить запись команд и обеспечить их унификацию в главах книги мы будем использовать обозначения R0, R1 и т. д. в ссылках на целочисленные регистры и обозначения F0, F1, ... в ссылках на регистры с плавающей запятой.

Как пример рассмотрим команду загрузки беззнакового байта:

```
LDUB  [R2+R3], R4
```

Команда осуществляет загрузку одного байта, хранящегося в памяти по адресу [R2] + [R3], в младшие 8 разрядов регистра R4, а его старшие 56 разрядов заполняет нулями. А вот команда загрузки слова со знаком

```
LDSW  [R2+2500], R4
```

считывает из памяти 32-разрядное слово, расположенное по адресу [R2] + 2500, расширяет его знак до 64 разрядов и помещает результат в регистр R4.

Арифметические и логические команды

Процессор SPARC имеет обычный набор арифметических и логических команд (табл. 8.1). Как вы уже знаете из раздела 8.4.2, команды должны устанавливать флаги условий только в тех случаях, когда известно, что последние будут проверяться следующей командой условного перехода. Это позволяет компилятору проявлять большую гибкость при реорганизации команд во избежание остановов конвейера. Именно в этом ключе разработан набор команд SPARC. Все арифметические и логические команды имеют по две версии: одна устанавливает флаги условий, а другая нет. Отличительным признаком команд первой категории является суффикс *cc*. Например, команды ADD, SUB, SMUL (умножение со знаком), OR и XOR не изменяют флаги условий, а команды ADDcc и SUBcc — изменяют.

В регистре R0 всегда содержится значение 0. Когда он используется в качестве результирующего операнда, результат выполнения команды уничтожается. Например, команда

```
SUBcc  R2, R3, R0
```

вычитает содержимое регистра R3 из регистра R2, устанавливает флаги условий и уничтожает результат вычитания. В итоге получаем команду сравнения, обладающую альтернативным синтаксисом:

```
CMP  R2, R3
```

Согласно терминологии SPARC, такая команда называется искусственной. Она не принадлежит к числу тех, которые распознаются аппаратным обеспечением, а используется лишь для удобства программистов. Впоследствии ассемблер заменяет ее командой SUBcc.

Для хранения флагов условий предназначен регистр CCR (Condition Code Register), содержащий наборы флагов *icc* и *xcc* для целочисленных и расширенных кодов условий. Каждый набор состоит из четырех флагов: N, Z, V и C. Команды, задающие флаги условий, такие как команда ADDcc, устанавливают разряды в обоих наборах. Флаги *icc* формируются на основе 64-разрядного результата команды, а флаги *xcc* — на основе младших 32 разрядов результата. Коды условий для операций с плавающей запятой хранятся в 64-разрядном регистре, или регистре состояния операций с плавающей запятой, — FSR (Floating Point Register).

Таблица 8.1. Некоторые команды процессора SPARC

Команда		Описание
ADD	R5, R6, R7	Целочисленное сложение: R7 ← [R5] + [R6]
ADDcc	R2, R3, R5	R5 ← [R2] + [R3], установка флагов кодов условий
SUB	R5, Imm, R7	Целочисленное вычитание: R7 ← [R5] − Imm (с расширенным знаком)
AND	R3, Imm, R5	Поразрядное И: R5 ← [R3] AND Imm (с расширенным знаком)
XOR	R3, R4, R5	Поразрядное исключающее ИЛИ: R5 ← [R3] XOR [R4]
FADDq	F4, F12, F16	Сложение чисел с плавающей запятой, четверная точность: F12 ← [F4] + [F12]
FSUBs	F2, F5, F7	Вычитание чисел с плавающей запятой, одинарная точность: F7 ← [F2] − [F5]
FDIVs	F5, F10, F18	Деление чисел с плавающей запятой, одинарная точность: F18 ← [F5] / [F10]
LDSW	R3, R5, R7	R7 ← 32-разрядное слово по адресу [R3] + R5 со знаком, расширенным до 64 разрядов
LDX	R3, R5, R7	R7 ← 64-разрядное расширенное слово по адресу [R3] + R5
LDUB	R4, Imm, R5	Загрузка беззнакового байта из памяти по адресу [R4] + Imm; байт загружается в младшие 8 разрядов регистра R5, все старшие разряды заполняются нулями
STW	R3, R6, R12	Сохранение слова из регистра R3 в памяти по адресу [R6] + [R12]
LDF	R5, R6, F3	Загрузка 32-разрядного слова по адресу [R5] + [R6] в регистр с плавающей запятой F3
LDDF	R5, R6, F8	Загрузка двойного слова (двух 32-разрядных слов) по адресу [R5] + [R6] в регистры с плавающей запятой F8 и F9
STF	F14, R6, Imm	Сохранение слова из регистра с плавающей запятой F14 в памяти по адресу [R6] + Imm
BLE	icc, Label	Проверка флагов icc и переход по адресу Label, если меньше или равно 0
BZ,pn	xcc, Label	Проверка флагов xcc и переход по адресу Label, если равно 0, переход предсказывается как невыполняемый
BGT,a,pt	icc, Label	Проверка 32-разрядных целочисленных кодов условий и переход по адресу Label, если больше 0, установка разряда Annul, переход предсказывается как выполняемый
FBNE,pn	Label	Проверка флагов состояния операций с плавающей запятой и переход по адресу Label, если не равно, установка разряда Annul в 0, переход предсказывается как невыполняемый

Команды перехода

На производительность компьютера в значительной степени влияет то, какой способ выбран для обработки переходов. Команды перехода SPARC имеют ряд особенностей и ориентированы на повышение производительности конвейерного процессора и оказание помощи компилятору в оптимизации программного кода.

В процессоре SPARC используется отложенный переход с одним слотом задержки (см. раздел 8.3.2). Команды перехода содержат разряд предсказания перехода, с помощью которого компилятор указывает аппаратному обеспечению, какого поведения следует ожидать от команды перехода. Кроме того, команды перехода содержат разряд Annul, обеспечивающий большую гибкость при обработке команды в слоте задержки. Эта команда выполняется всегда, но результаты не сохраняются до тех пор, пока не станет известно, осуществляется ли переход. Если да, команда завершается в слоте задержки, а результаты сохраняются. В противном случае команда аннулируется, если установлен разряд Annul, и завершается при несоблюдении этого условия.

Компилятор может поместить в слот задержки команду, которая нужна независимо от направления перехода. Это может быть команда, логически расположенная до перехода и допускающая перемещение в слот задержки. Для такой команды разряд Annul устанавливается в 0. Если же команда в слоте перехода должна быть вызвана только при осуществлении перехода, разряд Annul устанавливается в 1.

Команды условного перехода могут проверять флаги *icc*, *xcc* и FSR. Например, команда

```
BGT,a,pt    icc, Label
```

инициирует переход по адресу Label, если в результате выполнения предыдущей команды были установлены флаги *icc*, означающие «больше нуля». В этой команде и разряд Annul, и разряд предсказания перехода устанавливаются в 1. Команда

```
FBGT,a,pt    Label
```

производит те же действия, но проверяет флаги в регистре FSR. Если не задана ни опция pt (предсказывается выполнение перехода), ни опция pn (предсказывается невыполнение перехода), ассемблер устанавливает первую из них.

На рис. 8.22 приведен пример использования разрядов аннулирования и предсказания перехода в командах перехода. Это программный цикл, составляющий список из *n* 64-разрядных целых чисел. Предполагается, что количество элементов списка хранится по адресу LIST как 64-разрядное целое число; за ним в последовательных 64-разрядных фрагментах памяти расположены элементы списка. Мы также исходим из того, что список содержит хотя бы один элемент и что ранее в программе адрес LIST загружался в регистр R3.

На рис. 8.22, *a* показан цикл, написанный для неконвейерного процессора. Эффективное выполнение этого цикла процессором SPARC требует переупорядочения его команд таким образом, чтобы использовался слот задержки перехода. Команда ADD, следующая за меткой LOOPSTART, выполняется на каждой итерации цикла. Ни одна из дальнейших команд не зависит от результата ее работы.

Поэтому ее можно поместить в слот задержки, расположив после команды перехода в конце цикла (рис. 8.22, *б*). Поскольку команда ADD должна быть выполнена независимо от результата перехода, разряд Annul в команде перехода устанавливается в 0 (значение по умолчанию).

```
            LDX     R3, 0, R6          Загрузка количества элементов в списке
            OR      R0, R0, R4         Регистр R4 для отслеживания смещения
                                       в списке
            OR      R0, R0, R7         Очистка регистра R7, в котором будет
                                       накапливаться сумма
LOOPSTART   LDX     R3, R4, R5         Загрузка элемента списка в регистр R5
            ADD     R5, R7, R7         Добавление числа к сумме
            ADD     R4, 8, R4          Указатель на следующий элемент списка
            SUBcc   R6, 1, R6          Уменьшение значения регистра R6
                                       и установка флагов условий
            BG      xcc, LOOPSTART     Переход в начало цикла, если в списке
                                       еще остались элементы
NEXT        ...
```

а

```
            LDX     R3, 0, R6
            OR      R0, R0, R4
            OR      R0, R0, R7
LOOPSTART   LDX     R3, R4, R5
            ADD     R4, 8, R4
            SUBcc   R6, 1, R6
            BG,pt   xcc, LOOPSTART     Предсказание перехода, разряд Annul = 0
            ADD     R5, R7, R7
NEXT        ...
```

б

Рис. 8.22. Цикл сложения, демонстрирующий использование отложенного перехода и прогнозирование ветвления: исходный цикл программы (*а*); команды, реорганизованные для отложенного перехода (*б*)

Количество итераций цикла должно равняться количеству элементов в списке. Это означает, что до выхода из цикла переход будет выполнен неоднократно (за исключением простейшего случая, когда $n = 1$). Поэтому мы устанавливаем разряд предсказания перехода в команде BG, чтобы указать, что переход предсказывается как выполняемый.

Командами условного перехода являются не только те, которые проверяют флаги условий. Например, существует команда условного копирования MOVcc, копирующая данные из одного регистра в другой только в том случае, когда коды

условий удовлетворяют значению, заданному в суффиксе команды, *cc*. Рассмотрим две команды.

```
CMP     R5, R6
MOVle   icc, R5, R6
```

Команда MOVle копирует содержимое регистров R5 и R6, если флаги в *icc* соответствуют условию «меньше или равно» ($Z + (N \oplus V) = 1$). В результате меньшее из двух значений помещается в регистр R6. При отсутствии команды условного копирования ту же операцию позволили бы выполнить следующие команды:

```
        CMP     R5, R6
        BG      icc, GREATER
        MOVA    icc, R5, R6
GREATER ...
```

где MOVA — это команда безусловного перехода. Благодаря команде MOVle не только сокращается количество необходимых команд, но и предотвращается снижение производительности конвейерной обработки команд из-за перехода (что более важно).

У системы команд SPARC имеется множество других особенностей, способствующих повышению производительности суперскалярных процессоров со значительной степенью конвейеризации. С некоторыми мы познакомимся, изучая процессор UltraSPARC II.

8.7.2. UltraSPARC II

Основные компоненты процессора UltraSPARC II показаны на рис. 8.23. В процессоре используются два уровня кэш-памяти: внешний кэш (E-кэш) и два внутренних кэша (I-кэш для команд и D-кэш для данных). Контроллер внешнего кэша располагается на микросхеме процессора, как и аппаратное обеспечение для управления памятью. В блоке управления памятью имеется два буфера быстрого преобразования адресов: один предназначен для команд (iTLB), а другой — для данных (dTLB). Процессор взаимодействует с памятью и подсистемой ввода-вывода через системную шину.

Процессор располагает двумя блоками выполнения команд: один отведен для целочисленных операций, а другой — для операций с плавающей запятой. Каждый блок содержит набор регистров и два независимых конвейера для обработки команд. Таким образом, процессор может активизировать сразу четыре команды (две целочисленные и две с плавающей запятой), которые будут выполняться параллельно, каждая в своем конвейере. Если команды доступны постоянно и ни один из четырех конвейеров не приостановлен, на каждом такте может начинаться фаза выполнения четырех новых команд.

Блок упреждающей выборки и диспетчеризации PDU (Prefetch and Dispatch Unit) отвечает за бесперебойную поставку команд блокам выполнения. С этой целью он извлекает команды из памяти еще до того, как они потребуются, и помещает их во временный буфер, или буфер команд, играющий ту же роль, что и очередь команд на рис. 8.19.

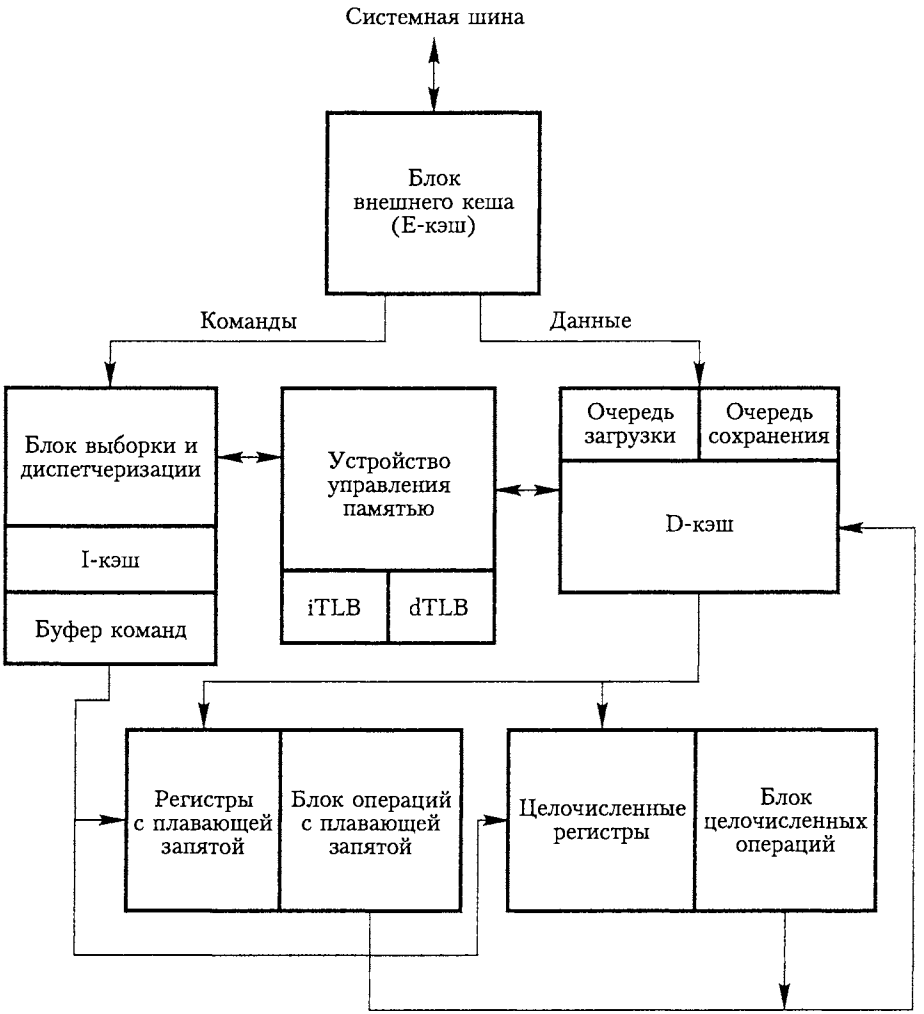


Рис. 8.23. Основные компоненты процессора UltraSPARC II

8.7.3. Структура конвейера

Процессор UltraSPARC II обладает 9-ступенчатым конвейером, структура которого представлена на рис. 8.24. На каждой ступени задача выполняется за один такт. Сначала мы рассмотрим работу конвейера в общем, а затем подробно обсудим каждую ступень.

Через первые три ступени конвейера проходят все команды. На первой (F) ступени команда извлекается из кэша, на второй (D) она частично декодируется, а на третьей (G) выбирается группа из четырех и менее команд для параллельного выполнения. Эти команды диспетчеризируются в блоки целочисленных операций и операций с плавающей запятой.

Каждый блок выполнения состоит из двух параллельных шестиступенчатых конвейеров. Четыре первые ступени предназначены для обработки поступающих команд, а две последние — для проверки исключений и сохранения результатов.

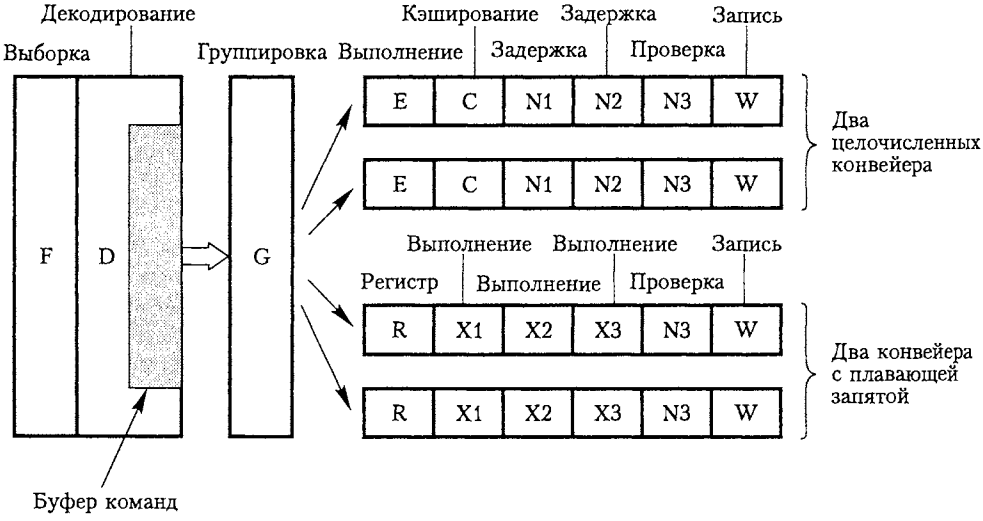


Рис. 8.24. Конвейер процессора UltraSPARC II

Выборка и декодирование команд

PDU выбирает из кэша четыре и менее команды, частично декодирует их и сохраняет результаты в буфере команд (максимум 12). Декодирование, выполняемое на этой ступени, позволяет PDU выявить команды перехода. Кроме того, PDU отмечает те особенности команд, которые могут способствовать ускорению принятия решений в конвейере.

Внутренний кэш команд (I-кэш) имеет объем 32 байта и может содержать до восьми команд. Вместе с каждой командой хранится ее виртуальный адрес, чтобы PDU мог быстро выбрать ее, не выполняя преобразование адресов. PDU способен обрабатывать по четыре команды за такт, если каждая группа не превышает границы кэша. Когда в кэше находится меньше четырех команд, PDU читает то, что есть.

Для предсказания переходов PDU использует алгоритм с четырьмя состояниями, подобный приведенному на рис. 8.15. Начальное состояние устанавливается в БП или МП в зависимости от значения разряда предсказания перехода. В PDU для каждых двух команд в кэше выделено два разряда. Туда записывается информация о состоянии алгоритма предсказания перехода. Эти разряды хранятся в связанных с командами тегах.

Каждым четырем командам в кэше отводится поле тега, или поле следующего адреса. Когда команда впервые выбирается для выполнения, PDU вычисляет целевой адрес перехода и записывает его в это поле. Оно используется для продолжения упреждающей выборки команд, чтобы PDU не приходилось каждый раз повторно вычислять целевой адрес перехода. Поскольку в каждой половине строки кэша имеется по одному полю следующего адреса, оно полноценно используется

только в том случае, если в каждой группе из четырех команд содержится хотя бы одна команда перехода.

Группировка

На третьей ступени конвейера, G, блок логики группировки выбирает группу из четырех и менее команд для параллельного выполнения и диспетчеризирует их в блоки целочисленных операций и операций с плавающей запятой. На рис. 8.25 показан порядок диспетчеризации короткой последовательности команд. Как будут сгруппированы команды в случае предсказания выполняемого или невыполняемого перехода, вы видите на рис. 8.25, *б* и 8.25, *в*. Обратите внимание на то, что команда в слоте задержки FCMP добавляется в выбранную группу в каждом случае. Она будет выполнена, но результаты останутся несохраненными до принятия решения о переходе. Если перехода не будет, результаты выполнения команды аннулируются, поскольку разряд Annul в команде перехода установлен в 1. Первые две команды каждой группы диспетчеризируются в целочисленный блок, а следующие две — в блок с плавающей запятой.

```
        ADDcc   R3, R4, R7   R7 ← [R3] + [R4]
                             Установка флагов условий
        BRZ,a   Label        Переход, если флаг условий равен нулю, установка
                             разряда Annul в 1
        FCMP    F1, F5       Операция с плавающей запятой: сравнение [F2] и [F5]
        FADD    F2, F3, F6   Операция с плавающей запятой: F6 ← [F2] + [F3]
        FMOVs   F3, F4       Перемещение операнда с одинарной точностью из F3
                             в F4
        :
Label   FSUB    F2, F3, F6   Операция с плавающей запятой: F6 ← [F2] – [F3]
        LDSW    R3, R4, R7   Загрузка слова, расположенного по адресу
                             [R3] + [R4], в [R7]
        :
```

а

```
        ADDcc   R3, R4, R7
        BRZ,a   Label
        FCMP    F1, F5
        FSUB    F2, F3, F6
```

б

```
        ADDcc   R3, R4, R7
        BRZ,a   Label
        FCMP    F1, F5
        FADD    F2, F3, F6
```

в

Рис. 8.25. Пример группировки команд: фрагмент программы (а); группировка команд при выполняемом переходе (*б*); группировка команд при невыполняемом переходе (*в*)

Логическая схема группировки отвечает за готовность к выполнению диспетчеризируемых команд. В частности, должны быть доступны все операнды команд

выбранной группы. В группу не может быть включена пара, в которой одна из команд зависит от другой. Исключением являются команды перехода.

Команды диспетчеризируются в том порядке, в котором они следуют в программе. Напомним, что команда перехода выполняется заблаговременно, то есть в блоке упреждающей выборки и декодирования выполняется предсказание перехода. Команды в буфере размещаются в соответствии с предсказанием перехода. Логический блок группировки анализирует команды в буфере и отбирает из начала очереди максимальное количество команд, удовлетворяющих условиям группировки. Давайте ознакомимся с некоторыми условиями отбора команд.

1. Команды диспетчеризируются только в порядке их следования. Если команду нельзя включить в группу, ни одна из следующих за ней команд в группу добавлена не будет.
2. Исходный операнд команды не может зависеть от результирующего операнда другой команды в той же группе. Но есть два исключения:
   - команда сохранения, записывающая содержимое регистра в память, может быть включена в одну группу с предшествующей командой, добавляющей данные в тот же регистр (это допускается потому, что, как вы вскоре убедитесь, команде сохранения данные требуются только на последней ступени);
   - команда перехода может быть сгруппирована с предшествующей командой, устанавливающей коды условий.
3. Две команды в группе не могут иметь один и тот же результирующий операнд, за исключением регистра R0. Например, команду LDSW на рис. 8.26, *а* нельзя помещать в одну группу с командой ADD.
4. В ряде случаев выполнение некоторых команд должно быть отложено на два или три такта. Так, команда условного копирования

   MOVRZ R1, R6, R7

   перемещает содержимое регистра R6 в регистр R7, если содержимое регистра R1 равно нулю. Для проверки содержимого регистра R1 этой команде требуется дополнительный такт. Поэтому команда, считывающая содержимое регистра R7, не может быть включена как в одну группу с ней, так и в следующую группу. На рис. 8.26, *б* приведен пример более ранней диспетчеризации такой команды.

Рис. 8.26. Задержки диспетчеризации: команды помещают результат в один и тот же регистр (*а*); задержка вызвана командой MOVR (*б*)

Когда логический блок группировки помещает команду в блок целочисленных вычислений, он также извлекает из целочисленного регистрового файла ее исходные операнды. Информация, необходимая для доступа к регистровому файлу, содержится в декодированных разрядах, помещенных в буфер команд блоком опережающей выборки и декодирования. Таким образом, к концу такта ступени G одна или две целочисленные команды готовы к ступени выполнения. Данные, считанные из регистрового файла, сохраняются в промежуточных буферах, как показано на рис. 8.27. Доступ к операндам в регистровом файле с плавающей запятой выполняется на ступени R после пересылки команды в блок операций с плавающей запятой.

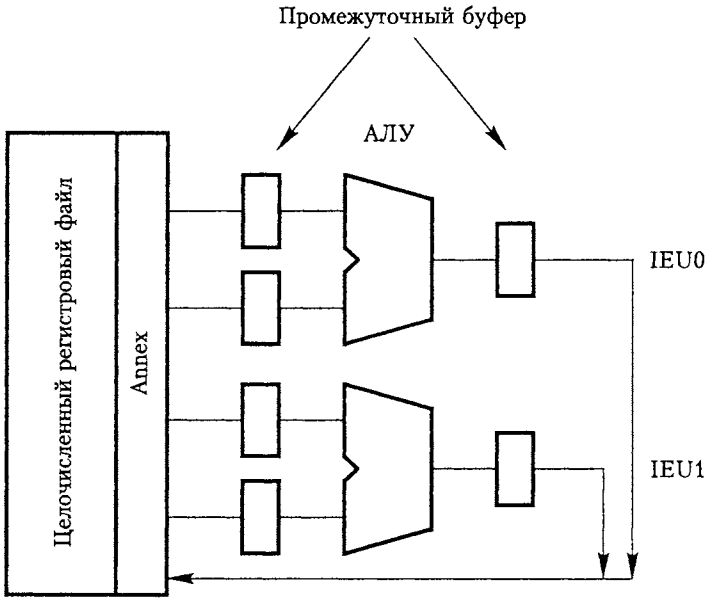


Рис. 8.27. Блок целочисленных операций

Блоки выполнения

Блок выполнения целочисленных операций содержит два сходных, но не одинаковых устройства — IEU0 и IEU1. Устройство IEU0 выполняет команды сдвига, а IEU1 генерирует коды условий. Команды, не содержащие таких операций, могут выполняться любым из устройств.

Большинство целочисленных команд выполняется АЛУ за один такт. В конвейере это ступень E. В момент завершения такта результат операции сохраняется в буфере. Взглянув на рис. 8.27, вы увидите его на выходе из АЛУ. В течение следующего такта, то есть на ступени C, содержимое буфера пересылается в регистровый файл, а точнее — в его часть, называемую Annex. Здесь находятся временные регистры, используемые для переименования регистров (см. раздел 8.6). Содержимое временного регистра передается в соответствующий постоянный регистр на ступени W.

На ступени C также генерируются коды условий. Это делается только для команд, явно включающих операцию установки флагов (например, ADDcc). Такие команды должны обрабатываться устройством IEU1.

Рассмотрим команду Icc, устанавливающую флаги кодов условий, и следующую за ней команду условного перехода BRcc, проверяющую эти флаги. Когда блок упреждающей выборки и декодирования встречает команду BRcc, результаты выполнения команды Icc могут быть еще недоступны. Блок PDU предсказывает результат выполнения команды перехода и на этом основании продолжает выборку команд. Позднее, когда команда Icc достигнет ступени C, коды условий будут сгенерированы и переданы в PDU на том же такте. PDU проверяет, правильно ли предсказан результат перехода, и, если да, продолжает выполнение программы. Если предсказание сделано неверно, содержимое конвейера и буфера команд стирается, и PDU приступает к выборке нужных команд. Отмена команд на этом этапе возможна потому, что они еще не достигли ступени конвейера W.

Неверное предсказание перехода чревато извлечением из памяти и даже частичным выполнением не тех команд. Описанная ситуация продемонстрирована на рис. 8.28. Предполагается, что логический блок группировки может диспетчеризировать по четыре команды на трех последовательных тактах. Команда Icc в начале первой группы устанавливает коды условий, проверяемые командой BRcc, которая следует далее. Проверка осуществляется, когда первая группа достигает ступени C. К этому моменту на ступень конвейера G передается третья группа команд, от I_9 до I_{12}. Если предсказание перехода ошибочно, девять команд — от I_4 до I_{12} — удаляются из конвейера (напомним, что команда I_3 в слоте задержки выполняется в любом случае). Кроме того, удаляются все команды, которые уже выбраны и загружены в буфер команд. Таким образом, в худшем случае может быть удалено до 21 команды.

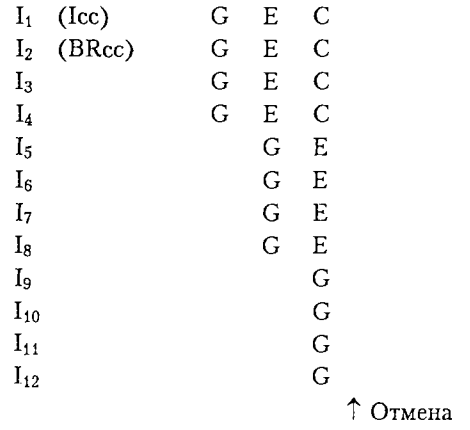


Рис. 8.28. Временная диаграмма работы конвейера в случае неверно спрогнозированного перехода

На ступенях конвейера N1 и N2 никаких операций не производится. Здесь обеспечивается задержка на два такта, необходимая для того, чтобы по длине целочисленный конвейер соответствовал конвейеру с плавающей запятой. Выполнение

целочисленных команд, не завершившееся на ступени C, продолжается на ступенях N1 и N2 (к таковым относится, в частности, команда деления). Если потребуется еще больше времени, дополнительные такты будут вставлены между ступенями N1 и N2. На ступень N2 команда попадает только во время последнего такта. Например, если для выполнения операции, заданной в команде, требуется 16 тактов, 12 из них будут вставлены после ступени N1.

Блок операций с плавающей запятой также содержит два независимых конвейера. Регистровые операнды выбираются на ступени R, операции с ними занимают до трех ступеней конвейера (от X1 до X3). Дополнительные такты, необходимые, скажем, для извлечения квадратного корня, вставляются между ступенями X2 и X3.

На ступени N3 процессор анализирует различные условия исключений, чтобы выяснить, не требуется ли прерывание. Наконец на ступени записи W результаты команды сохраняются по целевому адресу, будь то регистр или кэш данных. В любой момент до этой ступени команда может быть отменена, а результаты аннулированы, но после достижения ступени записи останов команды уже невозможен.

Блок загрузки и сохранения

Команда

LDUW R5,R6,R7

загружает в регистр R7 содержимое памяти по адресу [R5] + [R6]. Как и в случае других целочисленных команд, содержимое регистров R5 и R6 извлекается на ступени конвейера G. Однако, вместо одного из целочисленных блоков, команда и ее операнды пересылаются в блок загрузки и сохранения, структура которого показана на рис. 8.29. Работа этого блока начинается с вычисления суммы значений регистров R5 и R6 на ступени E для получения исполнительного адреса памяти. В результате получаем виртуальный адрес, который направляется в кэш данных. В то же время он передается в буфер dTLB для преобразования в физический адрес.

Данные хранятся в кэше в соответствии с их виртуальными адресами, что ускоряет обращение к ним, избавляя от необходимости преобразовывать адреса. Данные и соответствующие теги считываются из D-кэша на ступени C, а их физический адрес извлекается из dTLB. Тег, хранящийся в D-кэше, является частью физического адреса данных. На ступени N1 считанный из D-кэша тег сверяется с физическим адресом, полученным из dTLB. Если они совпадают, данные загружаются в регистр Annex для пересылки в целевой регистр команды на ступени W. В противном случае команда помещается в очередь загрузки/сохранения, где ожидает загрузки блока кэш-памяти из внешнего кэша в D-кэш.

Команда, помещенная в очередь загрузки/сохранения, более не считается находящейся в конвейере выполнения. Пока она дожидается своей очереди, выполнение других команд может завершиться, если, конечно, ни одна из них не содержит ссылку на регистр, в который должны быть прочитаны данные из памяти (в нашем примере — R7). Таким образом, очередь загрузки/сохранения изолирует конвейер от внешних операций доступа к данным с целью их независимого выполнения.

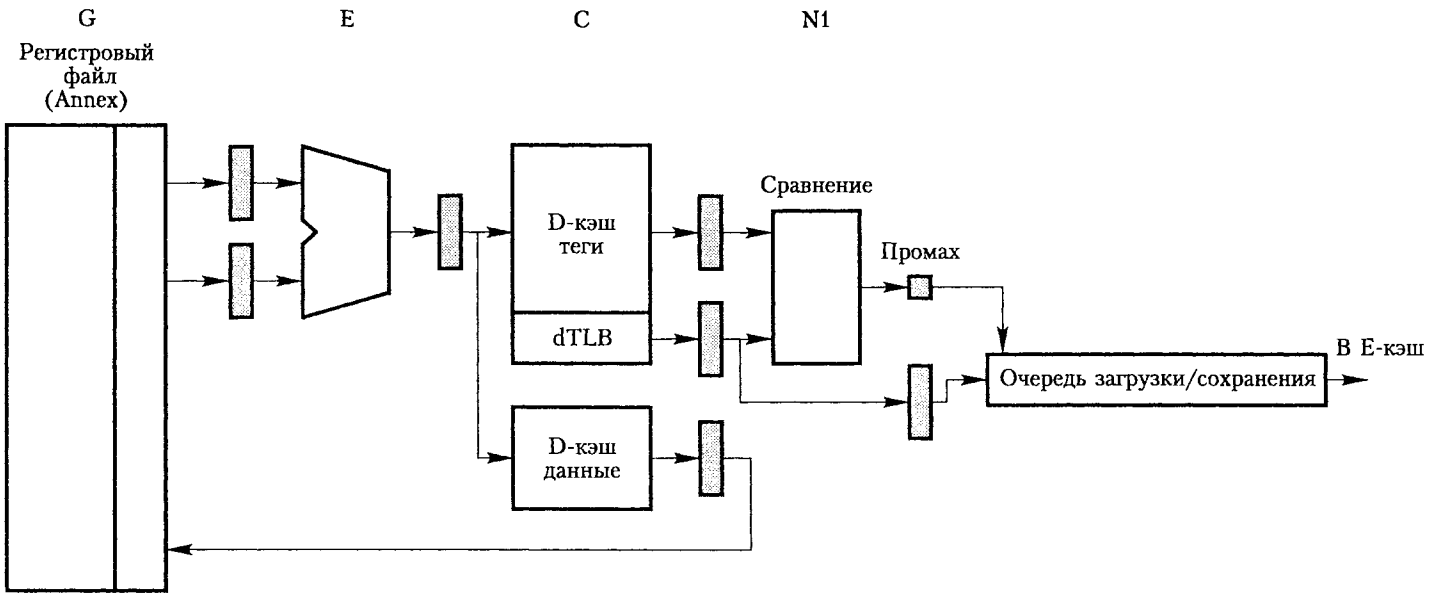


Рис. 8.29. Блок загрузки/сохранения

Поток выполнения

Давайте подробнее рассмотрим поток команд и данных внутри процессора UltraSPARC II, а также между ним, внешним кэшем и основной памятью. На рис. 8.30 показан поток команд и данных между основными функциональными блоками компьютера, представленными на рис. 8.23.

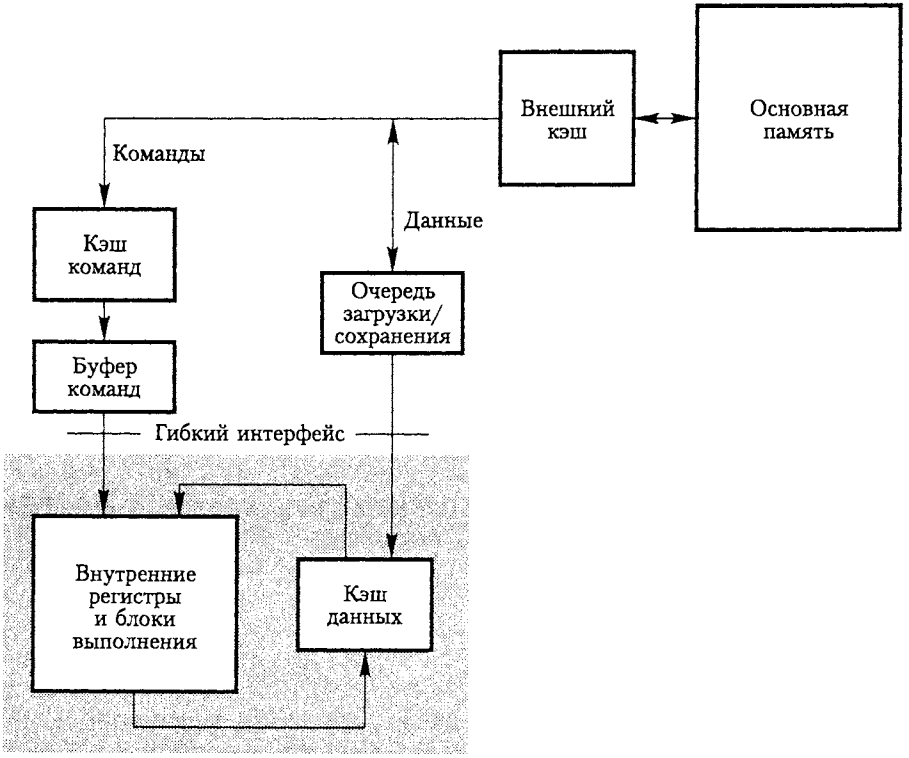


Рис. 8.30. Поток выполнения

Команды выбираются из I-кэша и загружаются в буфер команд, где их максимальное количество достигает 12. Оттуда по четыре за раз они передаются в блок «Внутренние регистры и блоки выполнения», в котором и выполняются. Средняя скорость, с которой PDU заполняет буфер команд, выше скорости их диспетчеризации в блоке группировки. Поэтому большую часть времени этот буфер заполнен. В случае отсутствия промахов при обращении к кэш-памяти и неверных предсказаний переходов внутренние блоки выполнения никогда не испытывают недостатка в командах. Поскольку операнды команд загрузки и сохранения, как правило, присутствуют в кэше, доступ к ним занимает один такт. Вот почему выполнение осуществляется без задержек.

Когда при обращении к кэшу команд обнаруживается промах, происходит задержка на несколько тактов, в течение которых блок данных загружается из внешнего кэша. В этот период блок группировки продолжает диспетчеризацию

команд, находящихся в буфере, пока тот не опустеет. Загрузка из внешнего кэша блока объемом 8 команд занимает три или четыре такта, что зависит от модели процессора. Примерно столько же времени требуется для диспетчеризации всех команд из заполненного буфера. (Следует отметить, что не всегда удается диспетчеризировать четыре команды на каждом такте.) Поэтому конвейер может работать бесперебойно, даже если в момент промаха кэш команд заполнен. А вот если нужного блока не окажется и во внешнем кэше, тогда придется обращаться к основной памяти, что потребует намного больше времени. В этом случае приостановка конвейера неизбежна.

Команда загрузки, послужившая причиной промаха при обращении к кэш-памяти, помещается в очередь загрузки/сохранения и ждет, когда поступят данные из внешнего кэша или основной памяти. При этом выполнение других команд может продолжаться при условии, что ими не используется целевой регистр операции загрузки. Таким образом, буфер команд и очередь загрузки/сохранения изолируют внутренний конвейер процессора от внешних пересылок данных. Они действуют как гибкие интерфейсы, позволяющие внутреннему высокоскоростному конвейеру продолжать работу во время выполнения более медленных операций пересылки внешних данных.

8.8. Производительность

В разделе 1.6 отмечалось, что время выполнения программы с динамическим количеством команд N рассчитывается по формуле

$$T = \frac{N \times S}{R}$$

где S — среднее количество тактов, затрачиваемых на выборку и выполнение одной команды, а R — тактовая частота процессора. В этой простой формуле предполагается, что команды выполняются одна за другой без наложения. Более полезным показателем производительности является *пропускная способность* процессора — количество команд, производимых за одну секунду. При последовательном выполнении пропускная способность P_s определяется по формуле

$$P_s = R/S$$

Далее мы проанализируем, как повышается пропускная способность процессора при конвейерной обработке команд. Давайте изберем иные принципы оценки производительности, чем те, которые предлагались в главе 1. Единственной реальной мерой производительности компьютера является общее время выполнения программы. Высокая пропускная способность сама по себе еще не является свидетельством хорошей производительности. Вот почему SPEC-коэффициенты, о которых рассказывается в главе 1, гораздо лучше характеризуют производительность компьютера, чем любые другие его параметры.

На рис. 8.2 показано, что 4-ступенчатый конвейер может вчетверо увеличить пропускную способность процессора. В общем случае n-ступенчатый конвейер потенциально повышает производительность в n раз. Получается, что, чем больше

значение n, тем выше производительность процессора. В этой связи возникают два вопроса:

- Какую пропускную способность процессора можно реализовать на практике?
- Каково наилучшее значение n?

Каждый раз, когда происходит останов конвейера, поток команд, обрабатываемых процессором, резко сокращается. Таким образом, производительность конвейера во многом зависит от таких факторов, как накладные расходы переходов и промахов при обращении к кэш-памяти. Сначала мы обсудим эти факторы, а затем вернемся к вопросу о количестве ступеней конвейера, которое можно реализовать на практике.

8.8.1. Конфликты по управлению

В предыдущих разделах было подробно рассказано о различных конфликтах в конвейере, а также об их последствиях. Сейчас мы возобновим разговор о промахах при обращении к кэш-памяти и накладных расходах переходов для того, чтобы оценить вызываемое ими снижение производительности компьютера.

Предположим, в процессоре используется 4-ступенчатый конвейер, показанный на рис. 8.2. Его тактовая частота, определяющая время, которое выделяется для каждого шага конвейера, соответствует времени выполнения самого длинного шага. Допустим, этим шагом является осуществление операции в АЛУ. Предположим также, что время, необходимое для сложения двух целых чисел, равно 2 нс и что тактовая частота процессора составляет 500 МГц. Тогда внутренняя кэш-память для команд и данных также должна проектироваться таким образом, чтобы время доступа к ней составляло 2 нс. В идеальных условиях пропускная способность P_p этого конвейерного процессора рассчитывается так:

$$P_p = R = 500 \text{ MIPS (миллионов команд в секунду)}$$

Оценивая влияние, которое оказывают на производительность промахи при обращении к кэшу, давайте воспользуемся параметрами, приведенными в разделе 5.6.2. Мы подсчитали, что накладные расходы промаха, M_p, в этой системе составляют 17 тактовов. Предположим, что T_1 — это временной интервал между двумя моментами завершения двух последовательно выполняемых команд. При последовательном выполнении $T_1 = S$. Однако при отсутствии конфликтов конвейерный процессор завершает выполнение одной команды на каждом такте, то есть $T_1 = 1$ такту. Промах при обращении к кэшу приводит к тому, что конвейер приостанавливается на количество тактов, равное накладным расходам промаха. Это означает, что для команды, при выполнении которой происходит промах, значение T_1 увеличивается на указанное количество тактов. Промахи возникают как при вызове команд, так и при чтении данных. Возьмем компьютер, в котором для команд и данных применяется общий кэш. Пусть процент команд со ссылками на данные в памяти равняется d. На сколько в среднем увеличивается значение T_1 из-за промахов при обращении к кэш-памяти, можно рассчитать по такой формуле:

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times M_p$$

Здесь h_i и h_d — это коэффициенты попаданий для команд и данных соответственно. Предположим, что к памяти обращается 30 % команд. При частоте попаданий 95 % для команд и 90 % для данных δ_{miss} определяется следующим образом:

$$\delta_{miss} = (0{,}05 + 0{,}3 \times 0{,}1) \times 17 = 1{,}36 \text{ такта}$$

С учетом этой задержки пропускная способность процессора

$$P_p = \frac{R}{T_1} = \frac{R}{1 + \delta_{miss}} = 0{,}42R$$

Обратите внимание на то, что, хотя R выражено в мегагерцах, мы получаем пропускную способность в миллионах команд в секунду. Если R = 500 МГц, то P_p равно 210 MIPS.

Давайте сравним это значение с пропускной способностью процессора без конвейерной обработки команд. Вот чему она равна:

$$P_s = \frac{R}{4 + \delta_{miss}} = 0{,}19R$$

При R = 500 МГц значение P_s равно 95 MIPS. Очевидно, что при конвейерной организации пропускная способность процессора значительно повышается. Однако, повысив производительность на 0,42/0,19 = 2,2, мы лишь немного превысим половину показателя в идеальном случае.

Сокращение накладных расходов промаха имеет особенно важное значение для конвейерных процессоров. Как рассказывалось в главе 5, его можно добиться за счет ввода вторичного кэша, который размещается между первичным кэшем, интегрированным в микросхему процессора, и внешней памятью. Предположим, что для пересылки из вторичного кэша блока объемом 8 слов требуется 10 нс. Если нужный блок данных не найден во вторичном кэше, накладные расходы составляют M_s = 5 тактов. Если же во вторичном кэше блока не окажется, полные накладные расходы M_p достигнут 17 тактов. Давайте определим, каково среднее увеличение значения T_1, когда частота попаданий для вторичного кэша составляет 94 %:

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times (h_s \times M_s + (1 - h_s) \times M_p) = 0{,}46 \text{ такта}$$

Пропускная способность процессора в этом случае равна 0,68R, или 340 MIPS. Эквивалентный неконвейерный процессор будет иметь пропускную способность 0,22R, или 110 MIPS. Таким образом, за счет введения конвейера производительность повышается на 0,68/0,22 = 3,1.

Значения 1,36 и 0,46 не выглядят многообещающе, ведь мы предполагали, что при каждом промахе задержка при обращении к памяти будет равна общему значению всех накладных расходов. Но так происходит в случае, когда откладывается выполнение команды, непосредственно следующей за той, которая послужила причиной промаха, и процессор ждет завершения обращения к памяти. Однако оптимизирующий компилятор по возможности отдаляет зависящие друг от друга команды, помещая между ними другие. Кроме того, если в процессоре существует

очередь команд, промах при выборке команды может иметь значительно менее негативные последствия, так как процессор продолжает выполнение находящихся в очереди команд.

8.8.2. Количество ступеней конвейера

Ввиду того что n-ступенчатый конвейер повышает производительность процессора в n раз, представляется целесообразным разделять процесс выполнения команд на как можно большее количество ступеней. Однако чем больше ступеней, тем выше вероятность остановов конвейера. Это связано с тем, что большинство команд выполняются параллельно, в связи с чем даже значительно отдаленные друг от друга команды, между которыми имеются зависимости, могут вызывать остановы конвейера. Как показано на рис. 8.9, возрастают накладные расходы переходов. По перечисленным причинам повышение производительности за счет увеличения количества ступеней оказывается не столь уж существенным.

Еще один немаловажный фактор, влияющий на производительность, — внутренние задержки при выполнении процессором базовых операций. Особого внимания заслуживает задержка в АЛУ. Во многих процессорах тактовая частота подбирается таким образом, чтобы операция АЛУ осуществлялась за один такт. Остальные операции разделяются на шаги, занимающие то же время, что и операция сложения. При этом АЛУ может иметь конвейерную организацию. Например, АЛУ процессора Compaq Alpha 21064 включает 2-ступенчатый конвейер, на каждой ступени которого работа выполняется за 5 нс. Во многих конвейерных процессорах используется от четырех до шести ступеней. В ряде процессоров процедура выполнения команды делится на меньшие шаги, используется большее количество ступеней конвейера и более высокая тактовая частота. Например, в процессоре UltraSPARC II применяется 9-ступенчатый конвейер, а в процессоре Intel Pentium Pro — 12-ступенчатый. Новейшая разработка Intel Pentium 4 содержит 20-ступенчатый конвейер и функционирует на тактовой частоте от 1,3 до 1,5 ГГц. Для ускорения работы на каждом такте выполняются две ступени конвейера.

8.9. Резюме

В этой главе вы познакомились с двумя важнейшими технологиями, позволяющими ускорить работу процессоров: конвейерной обработкой команд и суперскалярным выполнением команд. Применение технологии конвейерной обработки информации позволяет создавать процессоры с быстродействием, равным одной команде за такт. А процессоры с суперскалярной организацией способны выполнять по нескольку команд за такт.

Чтобы на деле добиться такой высокой производительности, нужно уделить особое внимание системе команд процессора, аппаратной структуре конвейера и разработке компилятора, которые, что немаловажно, тесно взаимосвязаны. Одним из ключевых элементов современных высокопроизводительных процессоров являются системы команд, ориентированные на конвейерное выполнение.

Упражнения

8.1. Рассмотрим следующую последовательность команд:

```
Add   #20,R0,R1
Mul   #3,R2,R3
Add   #3,R2,R3
Add   R0,R2,R5
```

Во всех этих командах результирующий операнд стоит на последнем месте. Изначально в регистрах R0 и R2 содержатся значения 2000 и 50 соответственно. Эти команды выполняются компьютером с 4-ступенчатым конвейером, подобным изображенному на рис. 8.2. Пусть первая команда выбирается на такте 1, а выборка команды занимает один такт.

а) Нарисуйте диаграмму, как на рис. 8.2, *а*. Опишите операции, выполняемые на каждой ступени конвейера в течение тактов с 1 по 4.

б) Покажите содержимое промежуточных буферов B1, B2 и B3 на тактах со 2 по 5.

8.2. Повторите упражнение 8.1 для следующей программы:

```
Add   #20,R0,R1
Mul   #3,R2,R3
Add   #$3A,R1,R4
Add   R0,R2,R5
```

8.3. Команда I_2 на рис. 8.6 откладывается из-за того, что она зависит от результатов выполнения команды I_1. Оккупировав ступень декодирования, команда I_2 блокирует команду I_3, которая, в свою очередь, блокирует команду I_4. Если предположить, что команды I_3 и I_4 не зависят ни от команды I_1, ни от команды I_2, а регистровый файл допускает параллельное выполнение двух шагов записи, как бы вы использовали дополнительные буферы, чтобы выполнить команды I_3 и I_4 раньше, чем показано на рис. 8.6? Создайте рисунок с новой последовательностью шагов.

8.4. Задержка-пузырь на рис. 8.6 возникает из-за того, что команда I_2 блокируется на ступени декодирования. В результате команды I_3 и I_4 задерживаются даже в том случае, если они не зависят от команды I_1 или I_2. Допустим, на ступени декодирования могут одновременно выполняться два шага. Докажите, что задержки можно избежать, если регистровый файл позволит параллельно совершать два шага записи.

8.5. На рис. 8.4 показана команда, отложенная из-за промаха при обращении к кэш-памяти. Создайте аналогичный рисунок для аппаратной организации, представленной на рис. 8.10. Предполагается, что в очереди команд может находиться до 4 команд, а блок выборки считывает из кэша по две команды.

8.6. Программный цикл завершается условным переходом в начало цикла. Как реализовать этот цикл для конвейерного процессора, в котором используется технология отложенных переходов с одним слотом задержки? При каких условиях в слот задержки можно будет поместить полезные команды?

8.7. В команде перехода процессора UltraSPARC II имеется разряд Annul. Если этот разряд установлен компилятором и переход не выполняется, команда из слота задержки удаляется из конвейера. Команда может быть удалена и в том случае, если переход выполняется. Каковы преимущества каждого из этих подходов?

8.8. Компьютер поддерживает один слот задержки. Команда в этом слоте выполняется независимо от предсказанного результата перехода, но если переход не выполняется, команда отменяется. Предложите эффективный способ реализации программных циклов для такого компьютера.

8.9. Перепишите подпрограмму сортировки для процессора SPARC, приведенную на рис. 2.34. Напомним, что в архитектуре SPARC определен один слот задержки с разрядом Annul и используется предсказание переходов. Постарайтесь заполнить слоты задержки полезными командами.

8.10. Рассмотрим такую команду:

IF A > B THEN действие 1 ELSE действие 2

Напишите соответствующую ей последовательность ассемблерных команд в двух вариантах: в первом используйте команды безусловного перехода, а во втором — команды условного перехода, подобные тем, которые имеются в системе процессора ARM. Если процессор содержит простой двухступенчатый конвейер, нарисуйте диаграмму, как на рис. 8.8, чтобы сравнить время выполнения двух ваших программ.

8.11. Канал продвижения данных, нарисованный на рис. 8.7 тонкими линиями, позволяет использовать содержимое регистра RSLT непосредственно в АЛУ. Результат этой операции записывается в регистр RSLT, заменяя его исходное содержимое. Какого типа регистр нужен для того, чтобы эта схема функционировала?

Рассмотрим две команды:

I_1: Add R1,R2,R3
I_2: Shift_left R1,R2,R3

Предположим, перед выполнением команды I1 регистры R1, R2, R3 и RSLT содержат значения 30, 100, 45 и 198 соответственно. Нарисуйте временную диаграмму для 4-ступенчатого конвейера, демонстрирующую состояние тактового сигнала и содержимое регистра RSLT на каждом такте. На основе этой диаграммы докажите, что продвижение операндов дает правильные результаты.

8.12. Напишите программу, аналогичную приведенной на рис. 2.37, для процессора, в котором доступ к памяти осуществляют только команды загрузки и сохранения. Выделите все имеющиеся в программе зависимости и продемонстрируйте, как ее оптимизировать для выполнения конвейерным процессором.

8.13. Команды перехода составляют 20 % количества команд программы. В компьютере применяется технология отложенных переходов с одним слотом задержки. Оцените, на сколько может быть повышена производительность, если компьютер использует 85 % слотов задержки.

8.14. У конвейерного процессора имеется два слота задержки. Оптимизирующий компилятор заполняет первый слот в 85 % случаев, а второй — в 20 % случаев. Определите в процентах, на сколько можно повысить производительность за счет такой оптимизации, если 20 % выполняемых команд программы составляют команды перехода?

8.15. В конвейерном процессоре применяется технология отложенных переходов. Вас просят порекомендовать один из двух вариантов архитектуры процессора. Согласно первому, процессор имеет 4-ступенчатый конвейер и один слот задержки, а в соответствии со вторым — 6-ступенчатый конвейер и два слота задержки. Сравните производительность этих двух архитектур с учетом накладных расходов на переходы. Предполагается, что 20 % выполняемых команд программы приходится на команды перехода, а оптимизирующему компилятору удается заполнить один слот задержки в 80 % случаев (вариант 1) и в 25 % случаев (вариант 2).

8.16. В процессоре используется механизм предсказания переходов, схема которого показана на рис. 8.15, б. Его начальным состоянием является БП или МП в зависимости от того, что указано в переходе. Поясните, как компилятор должен обрабатывать команды перехода, предназначенные для управления циклами «выполнять до» и «выполнять пока», и покажите, насколько подходит для каждого случая данный механизм предсказания переходов.

8.17. Предположим, что очередь на рис 8.10 может содержать до шести команд. Нарисуйте схему, которая подобна предложенной на рис. 8.11, для ситуации, когда на такте 1 очередь полна и блок выборки может одновременно считывать из кэша до двух команд. Когда очередь снова заполнится после выборки команды I_k?

8.18. Нарисуйте схему, которая подобна предложенной на рис. 8.11, для случая неверного предсказания перехода (рис. 8.14).

8.19. Как показано на рис. 8.16, для выполнения одной команды со сложным режимом адресации требуется столько же времени, как и для эквивалентной последовательности команд, рассчитанных на применение более простых режимов адресации. На использовании простых режимов адресации базируется философия RISC. Как разработать конвейер для обработки сложных режимов адресации? Укажите достоинства и недостатки этого подхода.

Глава 9

Встроенные системы

- ✦ Микроконтроллеры для встроенных систем
- ✦ Ограничения, накладываемые устройствами ввода-вывода
- ✦ Управление устройствами ввода-вывода с помощью программного кода на языке С
- ✦ Разработка систем на базе одной микросхемы

Как известно, компьютерные системы используются для самых разнообразных целей. Они имеют разную организацию, различные размеры и функции. Важнейшими факторами, которые должны учитываться при разработке любой встроенной системы, считаются ее производительность, надежность и стоимость. Любой из компьютеров, обрабатывающих большое количество графической информации, в том числе графические изображения и анимацию, должен иметь высокую производительность, чтобы обеспечивать работу в реальном масштабе времени. Причем очень важно добиться наивысшей производительности персонального компьютера или рабочей станции, не выходя за пределы допустимой рыночной стоимости. Но значительный рост производительности неизбежно влечет за собой удорожание компьютера. Наиболее мощные системы применяются для быстрого выполнения огромных объемов вычислений, в частности для моделирования сложных процессов, проектирования печатных плат, в различных САПР-системах и т. п. На персональном компьютере используемые для этой цели приложения выполнялись бы по многу часов. Поэтому на практике их обычно запускают на мощных серверах, имеющих не только гораздо более высокую производительность, но и более высокую цену. В главе 8 мы уже рассмотрели ряд вопросов, связанных с разработкой высокопроизводительных систем.

В то же время существует множество приложений, для функционирования которых высокопроизводительные процессоры не нужны. Так, в цифровых фотоаппаратах и видеокамерах, сотовых телефонах, видеотелефонах, торговых терминалах, кухонном оборудовании, автомобилях, во многих игрушках используется микропроцессорное управление. Для таких приложений важна не высокая производительность, а надежность. Кроме того, их электронные компоненты должны иметь небольшой размер и низкое энергопотребление. Всего этого можно достичь, поместив на одну микросхему не только процессор, но и некоторые интерфейсы ввода-вывода, схемы таймера и другие элементы — в таком случае вся компьютерная система управления будет состоять из минимального количества микросхем. Микропроцессорные микросхемы, содержащие интерфейсы ввода-вывода и память, обычно называют *микроконтроллерами*. Физическая система, управляющая

выполнением некоторых задач, а не просто производящая вычисления, называется *встроенной системой*. О таких системах и рассказывается в данной главе.

9.1. Примеры встроенных систем

В этом разделе мы рассмотрим три примера, демонстрирующих функции и возможности встроенных систем.

9.1.1. Микроволновая печь

Компьютерное управление часто применяется в бытовой технике. Типичным примером бытового прибора с компьютерным управлением может служить микроволновая печь. Основным ее компонентом является магнетрон — устройство, генерирующее электромагнитные микроволны, разогревающие пищу в ограниченном пространстве. Включенный магнетрон генерирует излучение максимальной мощности. Для снижения мощности он периодически выключается, а затем опять включается. Таким образом, управляя мощностью и общим временем нагрева, можно реализовать разные режимы приготовления пищи.

В спецификации микроволновой печи должны присутствовать опции, с помощью которых производятся, в частности, следующие действия:

- выбор (пользователем) мощности нагрева и времени приготовления пищи;
- определение (пользователем) последовательности шагов по приготовлению пищи;
- автоматический выбор параметров приготовления, когда пользователь указывает тип пищи (мясо, овощи, крупы и т. п.), а необходимая мощность нагрева и время вычисляются контроллером;
- автоматическое размораживание мяса с учетом его веса.

В микроволновой печи имеется дисплей, который может показывать:

- текущее время суток;
- время, оставшееся до окончания процесса приготовления блюда;
- различного рода информационные сообщения для пользователя.

Для оповещения о готовности блюда применяется звуковой сигнал. Кроме того, в печи имеются вытяжной вентилятор и освещение. Все эти функции и элементы могут контролироваться микропроцессором.

Средства ввода-вывода для взаимодействия с пользователям включают:

- клавиши цифровой клавиатуры от 0 до 9 и набор функциональных кнопок, таких как «Сброс», «Пуск», «Стоп», «Уровень мощности», «Авторазмораживание», «Автоприготовление», «Установка таймера», «Управление вытяжным вентилятором»; некоторые клавиши могут выполнять по нескольку функций, что позволяет уменьшить их общее количество (например, посредством нескольких последовательных нажатий кнопки управления вентилятором можно выбрать его скорость);
- визуальное средство вывода в форме жидкокристаллического дисплея (подобного 7-сегментному индикатору, описанному в приложении А;
- небольшой динамик, издающий сигналы заданной тональности.

Контроллер микроволновой печи можно реализовать в виде маленького компьютерного устройства на базе микропроцессора. Его вычислительные функции довольно просты: поддержка часового механизма, определение действий, соответствующих различным режимам приготовления пищи, выдача управляющих сигналов для включения и выключения таких устройств, как магнетрон и вентилятор, вывод информации на дисплей. Для выполнения этих задач может использоваться процессор относительно простой архитектуры. Реализующая их программа тоже будет небольшой. Причем она должна храниться в доступной только для чтения энергонезависимой памяти — лишь при этом условии она не будет исчезать при отключении питания. Кроме того, необходима небольшая по объему RAM, в которой бы производились вычисления и хранились пользовательские данные. Самым важным требованием к такой системе является наличие достаточных возможностей ввода-вывода для работы с кнопками, дисплеями и выходными управляющими сигналами.

При реализации такого контроллера важно найти решение, эффективное и с точки зрения стоимости. Для работы с внешними входными и выходными сигналами удобны параллельные порты ввода-вывода. Одна из возможных схем микроволновой печи показана на рис. 9.1.

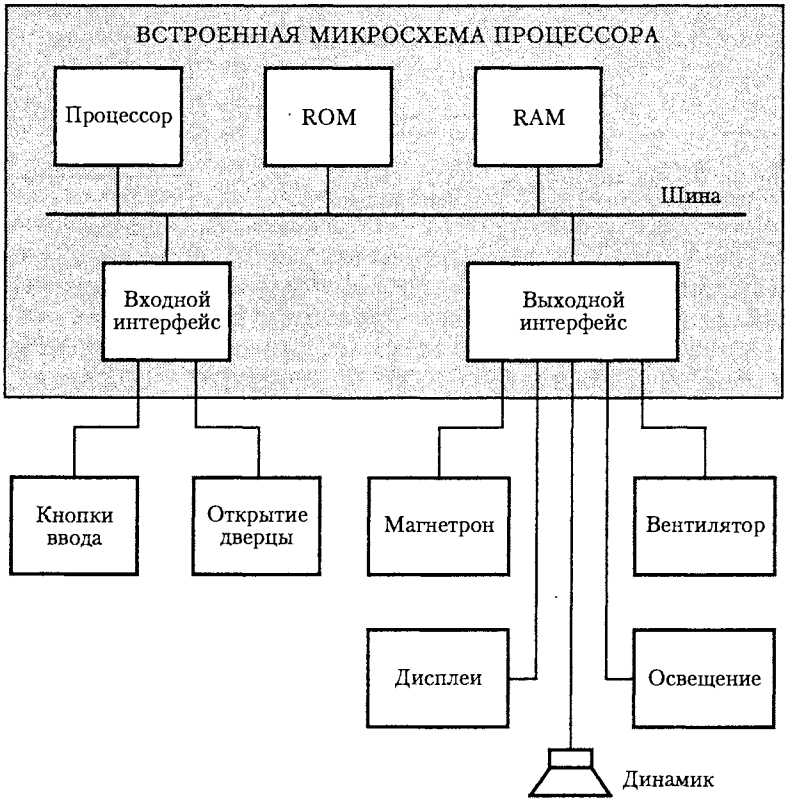


Рис. 9.1. Блок-схема микроволновой печи

Обратите внимание на то, как мало в ней аппаратных компонентов. Контроллер содержит простой процессор с небольшими по объему ROM и RAM, а также простые интерфейсы ввода-вывода для соединения с остальной частью системы. Почти всю эту схему можно реализовать на одной микросхеме СБИС.

9.1.2. Цифровой фотоаппарат

Цифровой фотоаппарат представляет собой наиболее удачный пример сложной встроенной системы, реализованной в очень маленьком модуле. Основные части фотоаппарата показаны на рис. 9.2.

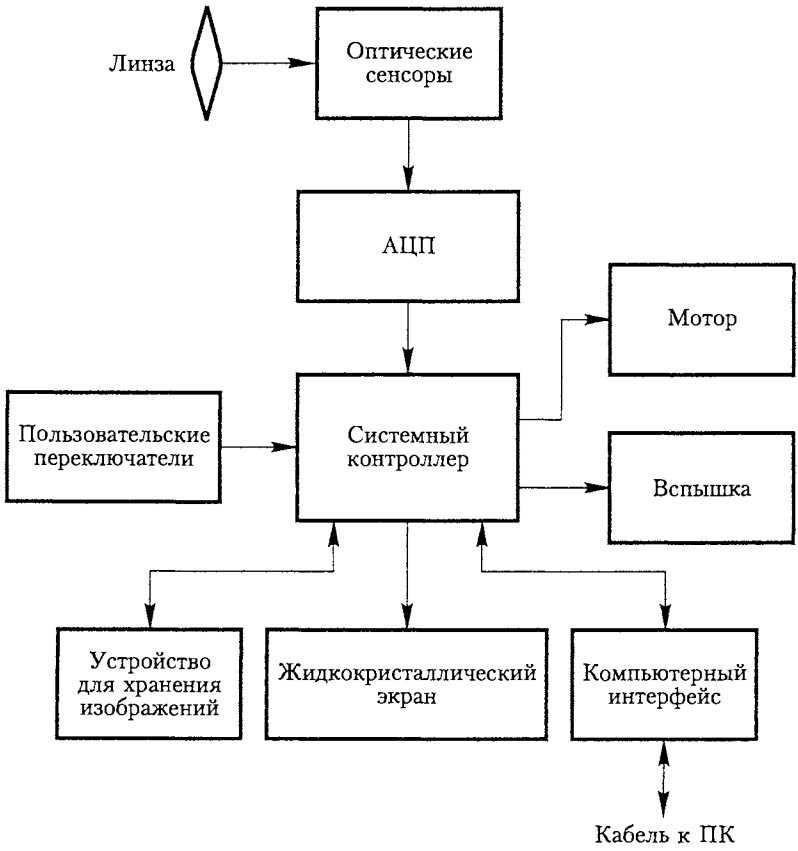


Рис. 9.2. Упрощенная блок-схема цифрового фотоаппарата

В фотоаппарате стандартной конструкции изображения отпечатываются на пленке. В цифровом аппарате фиксация изображения происходит с помощью массива оптических сенсоров, основанных на фотодиодах, которые преобразуют свет в электрический заряд. Интенсивность света определяет количество генерируемого заряда. В коммерческих фотоаппаратах используются сенсоры двух типов. К первому типу относятся ПЗС — приборы с зарядовой связью (Charge Coupled

Device, CCD). Они раньше других стали использоваться в цифровых видеокамерах и с тех пор неоднократно усовершенствовались с целью формирования более высококачественных изображений. Позже были разработаны сенсоры на основе КМОП-технологии — менее дорогие, но по качеству изображения уступающие сенсорам ПЗС-типа.

Каждому пикселу изображения соответствует аналоговый сигнал, генерируемый сенсорным элементом. От количества пикселов зависит качество записываемого изображения. Аналоговый сигнал преобразуется в дискретный с помощью *аналого-цифрового преобразователя* (*АЦП*). В полученном дискретном сигнале цвет и интенсивность каждого пиксела кодируются несколькими битами. Благодаря этому цифровое изображение можно обрабатывать посредством стандартных компьютерных схем.

Главным функциональным элементом фотоаппарата является системный контроллер. Этот блок содержит процессор, память (RAM и EEPROM), а также множество интерфейсных схем, необходимых для соединения различных частей системы. Процессор управляет всей работой камеры. Он обрабатывает сырые данные, полученные от аналого-цифрового преобразователя, генерирует изображения, представленные в стандартных форматах, которые подходят для использования в компьютерах, принтерах и устройствах отображения. Для несжатых изображений, как правило, используется формат TIFF, а для сжатых — JPEG.

Обрабатываемые изображения помещаются в устройство хранения изображений достаточно высокой емкости. В этой роли особенно популярны карты флэш-памяти, о которых рассказывалось в разделе 5.3.5. Кроме того, изображения могут храниться на дискетах и миниатюрных жестких дисках.

После обработки изображение можно вывести на имеющийся в камере жидкокристаллический дисплей (Liquid Crystal Display), или LCD-дисплей. Это дает возможность пользователю принять решение о целесообразности его сохранения. Количество фотографий, которое можно сохранить в фотоаппарате, зависит от емкости используемого в нем запоминающего устройства, а также от заданного качества изображений, то есть от количества пикселов в одном изображении.

Изображения без труда можно пересылать между фотоаппаратом и компьютером или принтером с помощью одного из стандартных компьютерных интерфейсов. Это может быть простой последовательный либо параллельный интерфейс или же соединительный разъем для подключения к стандартной шине, такой как PCI либо USB. Если в качестве запоминающего устройства используется флэш-карта, ее можно физически извлечь из фотоаппарата и вставить в соответствующий разъем компьютера.

Кроме прочего, системный контроллер генерирует сигналы, необходимые для управления двигателем (для фокусировки) и флэш-памятью. Некоторое количество входной информации поступает от пользователя через кнопки и переключатели.

Цифровому фотоаппарату требуется гораздо более мощный по сравнению с применяемым в микроволновой печи процессор, способный выполнять довольно сложные задачи по обработке изображений. Причем такой процессор не должен потреблять много энергии, поскольку фотоаппарат работает от батареек. Как правило, процессор потребляет даже меньше энергии, чем дисплей и флэш-память фотоаппарата.

9.1.3. Домашняя телеметрия

Компьютеры все шире используются в домашнем обиходе. Это и компьютеры общего назначения, и всевозможные встроенные системы для разнообразной техники. В разделе 9.1.1 мы рассмотрели микроконтроллер микроволновой печи. Подобные микроконтроллеры можно найти и во множестве других бытовых приборов, таких как стиральные машины, сушильные аппараты, электроплиты, кондиционеры, в отопительной системе. Еще одним наглядным примером может служить видеотелефон, в котором встроенный процессор кроме стандартных телефонных операций выполняет и множество иных полезных функций. В частности, он может применяться для удаленного доступа к другим устройствам, в том числе к компьютерному оборудованию.

Имея в своем распоряжении такой телефон, вы можете:

- взаимодействовать с управляемой компьютером системой сигнализации, установленной в вашем жилье;
- устанавливать оптимальную температуру для отопительной системы или кондиционера;
- задавать начальное время, время приготовления и температуру приготовления пищи, ранее помещенной в электро- или СВЧ-печь;
- считывать показания счетчиков электроэнергии, газа и воды, что особенно удобно, как вы понимаете, для обслуживающих компаний, сотрудникам которых теперь можно не обходить квартиры, с тем чтобы сделать это вручную.

Все эти функции можно реализовать при условии, что домашние устройства, с которыми взаимодействует телефон, также оборудованы микропроцессорами. В таком случае остается лишь обеспечить связь между микропроцессорами устройств и телефона. Причем организовать ее можно по-разному. Проще всего воспользоваться последовательным соединением, для реализации которого в микросхему контроллера должен быть интегрирован интерфейс UART, описанный в разделе 4.6.2. Процесс регистрации и обработки информации, поступающей от удаленной системы и используемой для контроля и управления оборудованием, часто называют *телеметрией*.

9.2. Процессорные микросхемы для встроенных систем

Микросхему, включающую процессор, некоторое количество памяти, интерфейс ввода-вывода и предназначенную для интеграции в различные устройства, часто называют *встраиваемым процессором*. Поскольку такие микросхемы выполняют важные управляющие функции и основаны на микропроцессорах, их также называют *микроконтроллерами*.

Общая структура встраиваемых процессорных микросхем достаточно гибкая, что позволяет использовать их в самых разных устройствах. Блок-схема типичной микросхемы показана на рис. 9.3. Основную ее часть составляет *процессорное*

ядро, которым может служить базовая версия одного из серийно выпускаемых микропроцессоров. Лучше всего использовать процессорную архитектуру, проверенную на практике, — в таком случае у вас не будет недостатка в средствах автоматизированного проектирования, хороших книгах, а также в специалистах, знания и опыт которых позволят быстро разрабатывать новые продукты.

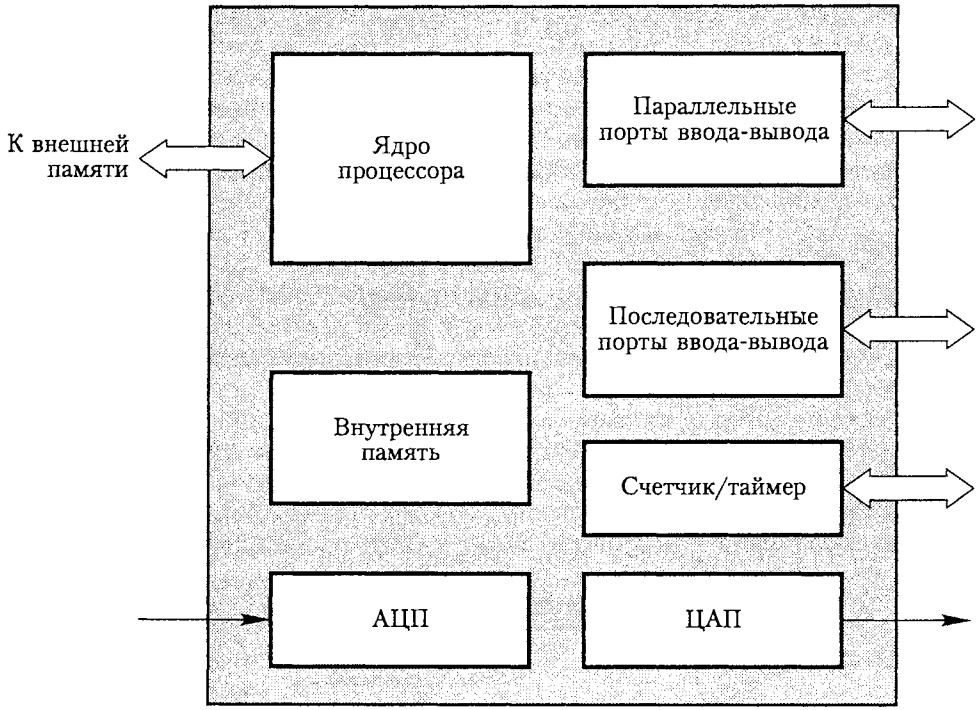


Рис. 9.3. Блок-схема встраиваемого процессора

Объем помещаемой на микросхему памяти должен быть достаточным для удовлетворения нужд ряда небольших приложений. Такая память должна состоять из двух частей: RAM — для хранения данных, которые изменяются в ходе вычислений, и ROM — для хранения программного обеспечения (во встроенных системах жесткого диска, как правило, не бывает). Для того чтобы ROM-память не была слишком дорогой, она должна быть программируемой. Наиболее популярными типами ROM-памяти для встроенных систем являются EEPROM и флэш-память.

Обычно процессорные микросхемы содержат несколько портов ввода-вывода для параллельного и последовательного интерфейсов. На их основе легко реализовать стандартные соединения для ввода-вывода. Во многих устройствах должны генерироваться периодические управляющие сигналы. Для этого во встраиваемые процессорные микросхемы включают схему таймера. А поскольку таймер способен отсчитывать тактовые импульсы, он может быть применен и для других задач, например, для подсчета количества импульсов на заданной входной линии.

В состав встроенных систем могут входить и некоторые аналоговые устройства. Для работы с такими устройствами необходимо обеспечить преобразование аналоговых сигналов в цифровые и наоборот. С этой целью во встраиваемый контроллер могут быть включены схемы ЦАП и АЦП.

Многие встраиваемые процессорные микросхемы выпускаются серийно. Наиболее популярны среди них семейства процессоров 68HC11, 683xx и MCF5xxx компании Motorola, процессор 8051 и семейство процессоров MCS-96 компании Intel, в которых используется ядро типа CISC, а также микроконтроллеры ARM с процессором типа RISC. Архитектура процессорного ядра для нас сейчас не имеет значения. Мы решили уделить основное внимание системным аспектам встраиваемых контроллеров, с тем чтобы показать на их примере, как описанные в предыдущих главах концепции можно объединить для разработки функционально полных встроенных компьютерных систем.

9.3. Простой микроконтроллер

В данном разделе речь пойдет об одном из возможных вариантов организации простого микроконтроллера, и на его примере мы продемонстрируем, как на практике используются некоторые типичные элементы и функции. Блок-схема нашего микроконтроллера приведена на рис. 9.4. Его микросхема включает процессорное ядро и некоторое количество памяти. А поскольку этой памяти может быть недостаточно для поддержки всех потенциально возможных приложений, для подключения дополнительной памяти микросхема имеет выводы процессорной шины.

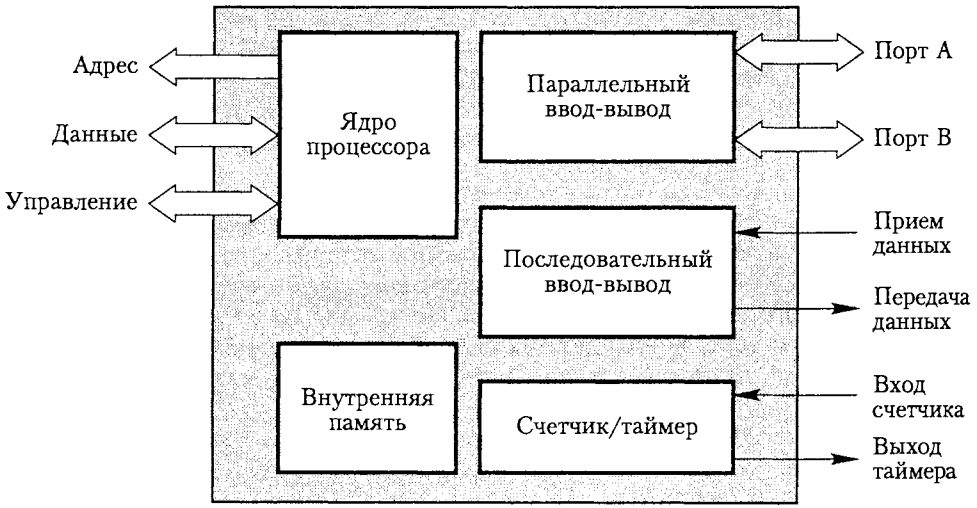


Рис. 9.4. Пример схемы микроконтроллера

Микроконтроллер содержит два 8-разрядных параллельных интерфейса, обозначенных как A и B, и один последовательный интерфейс. Кроме того, в нем

имеется 32-разрядный счетчик/таймер, который может, в частности, использоваться для генерирования сигналов внешних прерываний через программно задаваемые промежутки времени, реализации системного секундомера, подсчета количества импульсов на входной линии, генерирования прямоугольного выходного сигнала с переменным циклом и т. д.

9.3.1. Параллельные порты ввода-вывода

Параллельный интерфейс обеспечивает реализацию функций ввода-вывода подобно схеме, приведенной на рис. 4.34. Отдельные линии портов A и B могут использоваться для ввода или для вывода, что зависит от значения в регистре, который задает направление передачи данных. На рис. 9.5 приведена схема управления одним разрядом порта A. Соответствующий контакт этого порта PAi интерпретируется как входной, если триггер направления данных содержит 0. В этом случае в ответ на активизацию управляющего сигнала Read_Port на линию данных D_i помещается логическое значение с контакта PA_i. Мы не стали помещать на вход схемы запоминающий элемент (соответствует элементу DATAIN на рис. 4.34), поэтому процессор считывает данные прямо с контактов. Если триггер направления данных установлен в 1, контакт порта применяется для вывода данных. В этом случае по сигналу Write_Port на контакт помещается логическое значение из триггера данных. Поскольку каждому контакту соответствует свой бит направления данных, одни контакты могут быть запрограммированы как входные, а другие — как выходные.

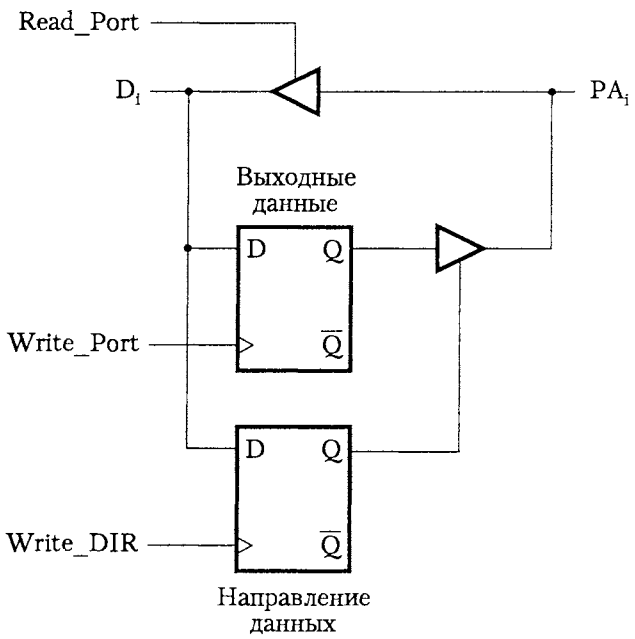


Рис. 9.5. Доступ к одному разряду порта A, показанного на рис. 9.4

В операциях по пересылке данных через порты A и B используются 8-разрядные регистры, показанные на рис. 9.6. На этом же рисунке приведены назначенные этим регистрам адреса, отображаемые в память. Мы выбрали их произвольным образом из верхней части 32-разрядного адресного пространства.

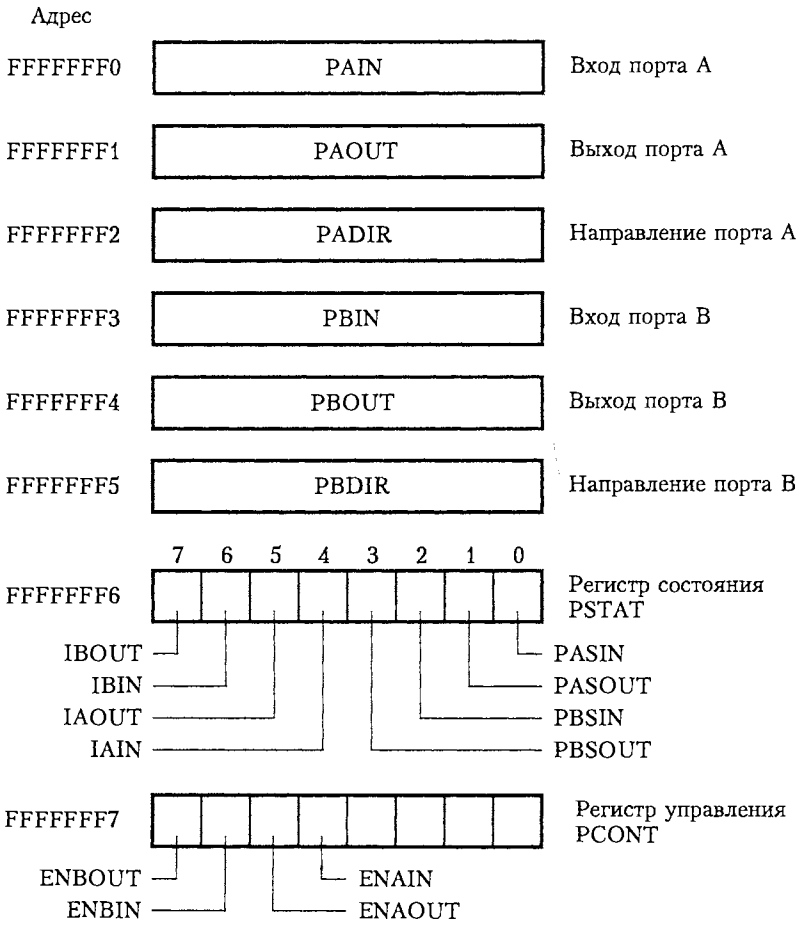


Рис. 9.6. Регистры параллельного интерфейса

Регистр состояния PSTAT содержит флаги состояний. Флаг PASIN устанавливается в 1, когда на контактах порта A появляются новые данные. Когда процессор принимает данные, считывая их из регистра PAIN, этот флаг очищается. После того как данные из регистра PAOUT отправляются подключенному к порту устройству, флаг PASOUT устанавливается в 1, указывая тем самым, что процессор может загружать в PAOUT новые данные. (Как будет рассказано ниже, пересылка данных устройству инициируется сигналами на управляющей линии.) Когда процессор записывает данные в PAOUT, флаг PASOUT очищается. Флаги PBSIN и PBSOUT выполняют аналогичные функции для порта B.

Кроме флагов состояний в регистре PSTAT содержатся четыре флага прерываний. Когда прерывание разрешено, установка соответствующего флага прерывания (например, IAIN) в 1 означает, что выполняется операция ввода-вывода. Биты разрешения прерываний хранятся в управляющем регистре PCONT. Для разрешения прерывания соответствующий разряд этого регистра устанавливается в 1. Например, если ENAIN = 1 (разрешено прерывание для порта A по входу) и PASIN = 1 (на контактах порта A имеются новые данные), тогда флаг прерывания IAIN (прерывание для порта A по входу) устанавливается в 1 и генерируется запрос прерывания. Таким образом,

$$IAIN = ENAIN \cdot PASIN$$

В системе используется единственный сигнал запроса прерывания, в ответ на который процессор анализирует флаги состояний и определяет его источник.

Информация, хранящаяся в регистрах состояния и управления, используется для проведения операций обмена данными с устройствами, подключенными к портам A и B. Порт A имеет две управляющие линии, CAIN и CAOUT, которые можно применить для обмена сигналами между интерфейсом и подключенным к нему устройством. Когда устройство направляет на контакты порта новые данные, оно сообщает об этом посредством активизации сигнала CAIN на один такт. Когда интерфейсная схема видит, что CAIN = 1, она устанавливает разряд состояния PASIN в 1. Позднее, после того как процессор считывает входные данные, этот разряд очищается. Последнее действие вызывает передачу интерфейсной схемой импульса по линии CAOUT, сообщающего устройству, что оно может отправить интерфейсу новые данные. Чтобы отправить данные через порт A, процессор записывает их в регистр PAOUT. В результате этого разряд PASOUT очищается и по линии CAOUT передается импульс, сообщающий устройству о поступлении новых данных. Когда устройство принимает эти данные, оно передает по линии CAIN подтверждающий импульс, а разряд PASOUT устанавливается в 1. Такой сигнальный механизм применяется при условии, что все контакты порта данных имеют одно и то же направление, то есть что порт используется либо только для ввода, либо только для вывода. Если же одни контакты назначаются в качестве входных, а другие — как выходные, то ни управляющие линии, ни регистры состояния и управления не будут содержать полезной информации.

9.3.2. Последовательный интерфейс

Последовательный интерфейс обеспечивает пересылку данных в соответствии со стандартом UART (Universal Asynchronous Receiver/Transmitter — универсальный асинхронный приемопередатчик), принцип действия которого показан на рис. 4.37. Для передачи и приема данных применяется, как следует из рис. 9.7, двойная буферизация. При обсуждении рис. 4.37 уже было отмечено, что эти буферы необходимы для обеспечения непрерывной пересылки данных.

На рис. 9.8 показаны адресуемые регистры последовательного интерфейса. Входные данные считываются из 8-разрядного буфера приема, а выходные данные загружаются в 8-разрядный буфер передачи. Регистр состояния SSTAT содержит информацию о текущем состоянии блоков приема и передачи данных.

Когда в буфере приема появляются новые данные, разряд $SSTAT_0$ устанавливается в 1. Разряд $SSTAT_1$ устанавливается в 1 при условии, что буфер передачи пуст и в него можно загружать новые данные. Эти разряды выполняют ту же функцию, что и флаги состояния SIN и SOUT, о которых говорилось в разделе 4.1. Если в процессе приема данных происходит ошибка, в 1 устанавливается разряд $SSTAT_2$. Ошибка может произойти, например, в том случае, если в буфер приема очередной символ будет записан до того, как процессор считает предыдущий. Кроме того, в регистре состояния содержатся флаги прерываний. Разряд $SSTAT_4$ устанавливается в 1, когда буфер приема полон и разрешены прерывания от приемника. Аналогичным образом, $SSTAT_5$ устанавливается в 1, когда буфер передачи пуст и разрешены прерывания от передатчика. Последовательный интерфейс генерирует прерывание, если 1 равен разряд $SSTAT_4$ или $SSTAT_5$. Кроме того, он генерирует прерывание в том случае, если $SSTAT_6 = 1$, а такое бывает при условии, что $SSTAT_2 = 1$ и разрешено прерывание из-за ошибки.

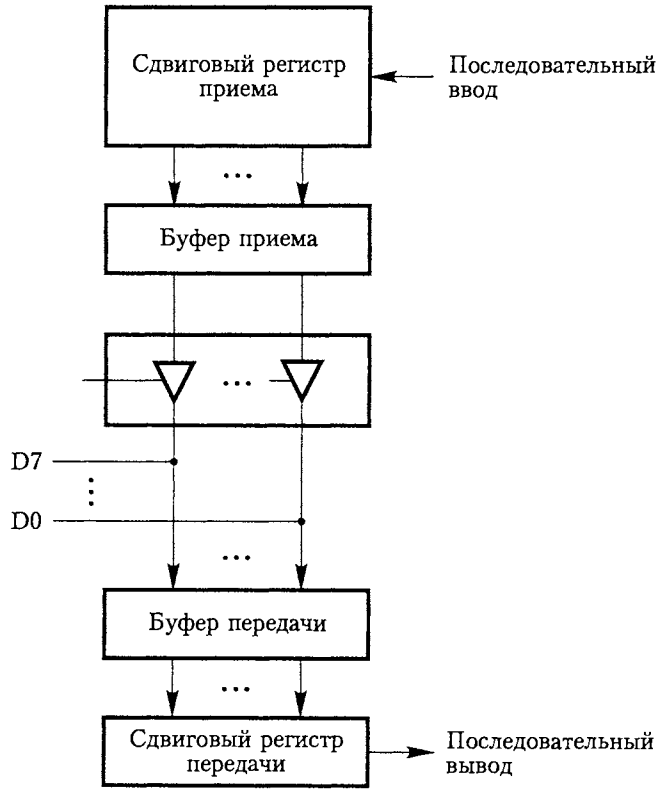


Рис. 9.7. Схема приема и передачи данных при использовании последовательного интерфейса

Регистр управления SCONT предназначен для хранения разряда разрешения прерывания. Установка разрядов $SCONT_{6-4}$ в 1 или 0 разрешает либо запрещает

соответствующие прерывания. Еще этот регистр указывает, как генерируется тактовый сигнал пересылки. Если $SCONT_0 = 0$, то указанный сигнал совпадает с тактовым сигналом процессора, но если $SCONT_0 = 1$, то тактовый сигнал пересылки генерируется путем деления сигнала процессора.

Еще один регистр последовательного интерфейса — это регистр делителя частоты DIV. Данный 32-разрядный регистр связан со схемой счетчика, которая делит системную тактовую частоту на его содержимое. Сигнал с результирующей тактовой частотой применяется для синхронизации последовательной пересылки данных. Деление выполняется очень просто. Значение регистра DIV пересылается в счетчик, который производит обратный отсчет, используя системный тактовый сигнал. Когда его значение становится равным нулю, в счетчик снова загружается значение из регистра DIV.

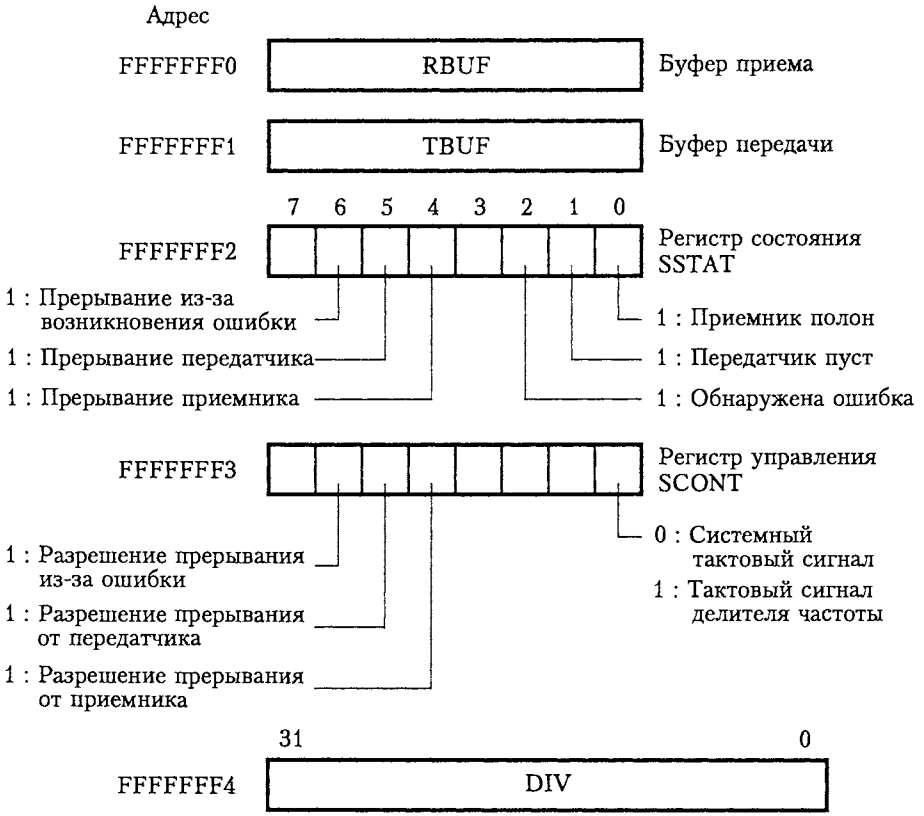


Рис. 9.8. Регистры последовательного интерфейса

9.3.3. Счетчик/таймер

32-разрядный счетчик с обратным отсчетом может использоваться и как счетчик, и как таймер. Основными его функциями являются установка начального значения

и уменьшение с определенной частотой значения в нем на единицу при использовании внутренних системных часов или внешнего тактового сигнала. Схема может быть запрограммирована на генерирование прерывания в тот момент, когда значение счетчика становится равным нулю. Регистры, связанные со схемой счетчика/таймера, показаны на рис. 9.9. В регистр CNTM загружается начальное значение, которое тут же будет переслано в схему счетчика. Текущее содержимое счетчика может быть прочитано по адресу памяти FFFFFFD4. Регистр управления CTCON используется для задания режима работы счетчика/таймера. С его помощью можно запускать и останавливать процесс отсчета, разрешать и запрещать прерывание по достижении значения 0. Регистр состояния CTSTAT отражает состояние схемы.

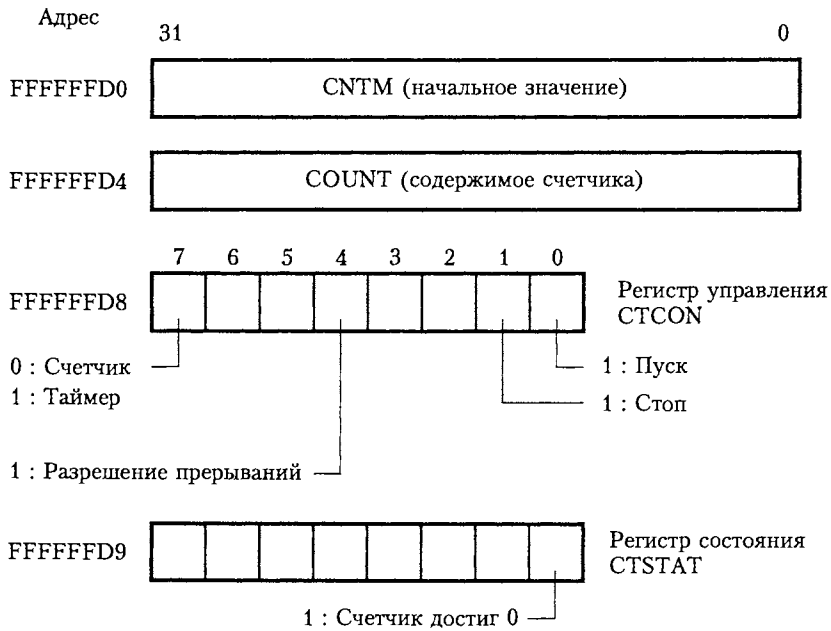


Рис. 9.9. Регистры счетчика/таймера

Режим счетчика

Режим счетчика выбирается путем установки разряда $CTCON_7$ в 0. Для того чтобы загрузить в счетчик начальное значение, его нужно записать в регистр CNTM. Процесс отсчета начинается, когда программа устанавливает разряд $CTCON_0$ в 1. Как только счетчик начинает отсчет, разряд $CTCON_0$ автоматически очищается. Значение счетчика уменьшается с помощью импульсов на линии Counter_in. Когда значение счетчика становится равным 0, флаг состояния $CTSTAT_0$ устанавливается в 1 и генерируется прерывание, если соответствующий разряд разрешения прерывания равен 1. Следующий тактовый импульс вызывает повторную загрузку в счетчик значения, которое хранится в регистре CNTM, и отсчет продолжается. Процесс отсчета останавливается путем установки в 1 разряда $CTCON_1$.

Режим таймера

Режим таймера задается установкой в 1 разряда $CTCON_7$. Этот режим предназначен для генерирования сигнала прямоугольной формы на выходной линии Timer_out (рис. 9.4). Процесс обратного отсчета времени начинается аналогично описанному выше процессу отсчета импульсов. Пока выполняется отсчет, значение на выходной линии остается постоянным. По достижении нуля в счетчик автоматически загружается начальное значение и выходной сигнал инвертируется. Таким образом, период выходного сигнала равен удвоенному периоду управляющего тактового сигнала, умноженному на начальное значение счетчика. В режиме таймера значение счетчика уменьшается на 1 по системному тактовому сигналу.

9.3.4. Механизм управления прерываниями

В микроконтроллере имеется две линии запроса прерывания, IRQ и XRQ. Линия IRQ предназначена для прерываний, генерируемых интерфейсами ввода-вывода внутри микроконтроллера, а линия XRQ — для прерываний, генерируемых внешними устройствами. Когда процессор обнаруживает, что линия IRQ активизирована, он путем опроса определяет источник (или источники) запроса прерывания, проверяя с этой целью флаги в регистрах состояния PSTAT, SSTAT и CTSTAT. Прерывание XRQ имеет более высокий приоритет, чем IRQ.

В регистре состояния процессора имеется два разряда для разрешения прерываний. Прерывания IRQ возможны в том случае, если $PSR_6 = 1$, а прерывания XRQ — если $PSR_7 = 1$. После обработки поступившего прерывания процессор запрещает дальнейшие прерывания с тем же уровнем приоритета, для чего перед вызовом программы обработки таковых очищает соответствующий разряд регистра PSR. При этом он использует векторную схему прерываний и хранит векторы IRQ и XRQ в памяти по адресам $24 и $28 соответственно. Каждый вектор содержит адрес первой команды программы, обрабатывающей прерывания определенного типа.

В нашем процессоре имеется регистр связи LR, который применяется для связывания подпрограмм (о нем уже рассказывалось в разделе 2.9). При выполнении команды вызова подпрограммы Call, до перехода к первой команде этой подпрограммы, обновленное содержимое счетчика команд PC (то есть адрес возврата из подпрограммы) загружается в регистр LR. То же самое происходит и при обработке прерывания. Однако в этом случае наряду с сохранением адреса возврата в регистре LR в регистре процессора IPSR сохраняется содержимое регистра состояния процессора PSR.

Возврат из подпрограммы выполняется с помощью команды ReturnS, которая пересылает содержимое регистра LR в PC. Возврат из прерывания выполняется с применением команды ReturnI, которая пересылает содержимое регистров LR и IPSR в PC и PSR соответственно. Поскольку в процессоре имеется только один регистр LR и один регистр IPSR, для реализации вложенных прерываний содержимое этих регистров можно сохранять в стеке посредством команд программы обработки прерывания. Подобный подход применяется в процессоре ARM, описанном в разделе 4.3.1.

9.4. Программирование

Теперь вы имеете представление об аппаратном обеспечении микроконтроллера, и мы можем перейти к вопросам программирования. Программы для микроконтроллера можно писать как на языке ассемблера, так и на языках высокого уровня. Для большинства приложений второй вариант гораздо удобнее, поскольку программы на языках высокого уровня проще и разрабатывать и сопровождать. Далее в этом разделе приведено несколько примеров программ на обоих языках. Это очень простые примеры, единственная цель которых — проиллюстрировать возможные подходы к программированию микроконтроллеров. В разделе 9.5 дан более сложный пример полного приложения. В качестве языка высокого уровня мы выбрали язык C.

Рассмотрим работу микроконтроллера, который предназначен для пересылки 8-разрядных символов из последовательного источника в параллельный приемник. Источник соединен с последовательным интерфейсом, а приемник — с параллельным портом A. Индикатором наличия символа в буфере приема является соответствующий разряд регистра состояния, то есть на присутствие символа указывает состояние $SSTAT_0 = 1$. Параллельный порт должен быть сконфигурирован для вывода, поэтому все разряды регистра направления данных устанавливаются в 1 ($PADIR_{7-0} = \$FF$). Мы предполагаем, что выходное устройство, получающее символы через порт A, функционирует быстрее источника, передающего символы через последовательный интерфейс. Следовательно, нет необходимости опрашивать порт A, чтобы выяснить, готов ли он к получению очередного символа. Для начала мы покажем, как выполнить требуемую пересылку с использованием опроса, а затем реализуем эту же задачу с помощью прерываний.

9.4.1. Использование опроса

Технология опроса заключается в периодической проверке флагов состояния, указывающих, получен ли новый символ (см. раздел 2.7). В нашем примере будет опрашиваться разряд $SSTAT_0$.

Программа на языке ассемблера

На рис. 9.10 показано, как поставленную задачу можно запрограммировать с применением языка ассемблера. Мы используем универсальный формат символов, описанный в главе 2. Ссылки на регистры микроконтроллера задаются посредством символических имен, связанных с адресами регистров. Программный цикл постоянно проверяет состояние разряда $SSTAT_0$. Каждый раз, когда $SSTAT_0 = 1$, символ из буфера приема RBUF перемещается в порт A. Напомним, что в результате операции четния из регистра RBUF разряд $SSTAT_0$ автоматически очищается. Если $PSTAT_1 = 1$, символ записывается в регистр PAOUT.

Для пересылки непрерывного потока символов в нашей программе используется бесконечный цикл (рис. 9.10). В реальном приложении программист едва ли станет применять такой цикл, поскольку приложение должно выполнять и другие задачи. Мы же выбрали этот способ лишь для упрощения примера.

```
RBUF    EQU     $FFFFFFE0   Приемный буфер
SSTAT   EQU     $FFFFFFE2   Регистр состояния последовательного
                            интерфейса
PAOUT   EQU     $FFFFFFF1   Выходные данные порта A
PADIR   EQU     $FFFFFFF2   Регистр направления порта A

*Инициализация
        ORIGIN  $1000
        MoveByte #$FF,PADIR Конфигурирование порта A в качестве
                            выходного
*Передача символов
LOOP    TestBit #0,SSTAT    Проверка того, готов ли новый символ
        Branch=0 LOOP
        MoveByte RBUF,PAOUT Пересылка символа в порт A
        Branch  LOOP
```

Рис. 9.10. Программа на унифицированном языке ассемблера, выполняющая пересылку символов с применением опроса

Программа на языке C

В программе на языке C отображаемый в память адрес ввода-вывода можно представить с помощью переменной-указателя. Этот адрес будет служить значением переменной. Если содержимое памяти по данному адресу интерпретируется как символ, переменная должна быть объявлена как указатель на данные типа char. Это будет означать, что по хранящемуся в переменной адресу располагается один байт данных — именно такой размер имеют наши регистры ввода-вывода. Для работы с содержимым регистров удобно использовать шестнадцатеричную форму.

На рис. 9.11 приведена программа на языке C, выполняющая ту же задачу, что и представленная выше программа на языке ассемблера. Инструкции define в начале этой программы связывают адреса с символическими именами указателей. Эти инструкции делают то же самое, что и команды EQU на рис. 9.10. Кроме того, они позволяют препроцессору компилятора языка C заменить символические имена в программе их реальными значениями. В результате откомпилированный код будет подобен коду на рис. 9.10.

Обратите внимание, что указатели RBUF и SSTAT объявлены как volatile. Объясняется это тем, что программа только считывает содержимое по заданным адресам, но никогда его не записывает. Оптимизирующий компилятор может удалить инструкции программы, содержащие переменные, значения которых никогда не изменяются. А поскольку содержимое регистров RBUF и SSTAT изменяется вне программы, важно проинформировать об этом компилятор. Переменные, объявленные как volatile, компилятор не удаляет.

В программе, которая представлена на рис. 9.12, для решения поставленной задачи применен другой подход. Вместо того чтобы определять указатели на регистры ввода-вывода в качестве констант, как это сделано в программе на рис. 9.11, мы объявляем их в качестве переменных, указывающих на символьные данные. Поэтому такие обозначения, как RBUF и PAOUT, определяют адреса памяти, по

которым содержатся реальные адреса регистров ввода-вывода. В этом случае откомпилированный код может быть таким, как показано на рис. 9.13. Для доступа к конкретному регистру ввода-вывода его адрес загружается в регистр процессора, после чего к регистру ввода-вывода можно обращаться с использованием режима косвенной регистровой адресации.

```
/*Определение адресов регистров/*
#define RBUF (volatile char *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2

void main ()

{
  /*Инициализация параллельного порта*/
  *PADIR = 0xFF;                    /*Конфигурирование порта A как выходного/*

  /*Пересылка символов/*
  while (1) {                       /*Бесконечный цикл/*
    while ((*SSTAT & 0x1)==0);      /*Ожидание нового символа/*
    *PAOUT = *RBUF;                 /*Пересылка символа в порт A/*
  }
}
```

Рис. 9.11. Программа на языке C, выполняющая пересылку символов с использованием опроса

```
/*Определение адресов регистров/*
volatile char *RBUF = (char *) 0xFFFFFFE0;
volatile char *SSTAT = (char *) 0xFFFFFFE2;
char *PAOUT = (char *) 0xFFFFFFF1;
char *PADIR = (char *) 0xFFFFFFF2;

void main ()

{
  /*Инициализация параллельного порта/*
  *PADIR = 0xFF;                    /*Конфигурирование порта A как выходного/*

  /*Пересылка символов/*
  while (1) {                       /*Бесконечный цикл/*
    while ((*SSTAT & 0x1) == 0); /*Ожидание нового символа/*
    *PAOUT = *RBUF;                 /*Пересылка символа в порт A/*
  }
}
```

Рис. 9.12. Альтернативная программа на языке C, выполняющая пересылку символов с использованием опроса

На рис. 9.13 показана только часть кода программы. Компилятор заменяет символические имена RBUF, SSTAT, PAOUT и PADIR соответствующими значениями адресов памяти. Машинный код программы, приведенной на рис. 9.12, имеет больший объем, чем машинный код программы с рис. 9.11. В последующих примерах мы будем определять указатели так же, как это сделано на рис. 9.11. Инструкции define подчеркивают, что определяемые ими адреса ввода-вывода являются постоянными, то есть в ходе выполнения программы они не изменяются.

```
        Move        PADIR,R0
        MoveByte    #$FF,(R0)
LOOP    Move        SSTAT,R0
        TestBit     #0,(R0)
        Branch=0    LOOP
        Move        RBUF,R0
        Move        PAOUT,R1
        Move        (R0), (R1)
        Branch      LOOP
```

Рис. 9.13. Вариант откомпилированного кода для программы, приведенной на рис. 9.12 (фрагмент)

В программах данной главы, написанных на языке С, мы непосредственно загружаем в регистры значения, необходимые для выполнения некоторых действий. Например, на рис. 9.11 инструкция

```
*PADIR = 0xFF;
```

устанавливает все восемь разрядов регистра PADIR в 1 и тем самым конфигурирует порт А как выходной. В программах на языке С обычно поступают иначе: объявляют для таких констант имена, которые затем используются на протяжении всей программы. Мы решили задавать константы явно лишь для того, чтобы максимально упростить программы и сделать их как можно короче — в таком случае вам будет легче сопоставлять полученные значения со спецификациями регистров ввода-вывода, приведенными на рис. 9.6–9.9.

9.4.2. Использование прерываний

Вместо того чтобы опрашивать регистр $SSTAT_0$ при необходимости узнать, не поступил ли новый символ, можно настроить интерфейс ввода-вывода таким образом, чтобы он сам генерировал запрос прерывания, когда $SSTAT_0 = 1$. Для этого соответствующий разряд разрешения прерываний в регистре SCONT, а именно $SCONT_4$, должен быть установлен в 1. Необходимо также разрешить прерывания IRQ в процессоре, для чего нужно установить в 1 разряд PSR_6. С этой целью можно загрузить в регистр PSR значение $40 и тем самым запретить прерывания XRQ. Адрес программы обработки прерывания нужно поместить в память по адресу $24.

Программа на языке ассемблера

На рис. 9.14 приведена программа на языке ассемблера, выполняющая пересылку символа с использованием прерываний. Запрос прерывания генерируется, когда $SSTAT_0 = 1$. В ответ программа обработки прерываний пересылает символ из RBUF в PAOUT. Чтение содержимого буфера приема RBUF вызывает очистку разряда $SSTAT_0$. Обратите внимание, что в программе нет ссылок на регистр состояния SSTAT. Как и в предыдущем примере, с целью упрощения программы ожидание нового символа выполняется с помощью бесконечного цикла.

```
RBUF     EQU       $FFFFFFE0     Приемный буфер
SSTAT    EQU       $FFFFFFE3     Регистр состояния последовательного
                                 интерфейса
PAOUT    EQU       $FFFFFFF1     Выходные данные порта A
PADIR    EQU       $FFFFFFF2     Регистр направления порта A
*Инициализация
         ORIGIN    $1000
         MoveByte  #$FF,PADIR    Конфигурирование порта A в качестве
                                 выходного
         Move      #$INTSERV,$24 Задание вектора прерывания
         Move      #$40,PSR      Процессор отвечает на прерывания IRQ
         MoveByte  #$10,SCONT    Выдача разрешения на прерывание,
                                 поступившее от приемника
*Цикл пересылки
LOOP     Branch    LOOP          Бесконечный цикл ожидания
*Программа обработки прерываний
INTSERV  MoveByte  RBUF,PAOUT    Пересылка символа в порт A
         Return                  Возврат из процедуры обработки
                                 прерывания
```

Рис. 9.14. Программа на унифицированном языке ассемблера, выполняющая пересылку символов с использованием прерываний

Программа на языке C

Для того чтобы написать на языке C программу с использованием прерываний, нам нужно решить два вопроса:

- как будет осуществляться доступ к регистрам процессора;
- какой должна быть программа обработки прерываний.

Чтобы получить возможность использовать прерывания, нужно соответствующим образом установить разряды управления прерываниями в регистре состояния процессора. В этом случае программа должна произвести запись некоторого значения в регистр PSR. В языках высокого уровня, в том числе и в C, после компиляции программы переменные представляют собой адреса памяти. Поэтому

с отображаемыми в память регистрами ввода-вывода можно работать так, как это делается в программах, представленных на рис. 9.11 и 9.12. Однако у регистров процессора, и в частности у регистра состояния PSR, нет адресов памяти. Вот почему для получения доступа к ним приходится прямо в программу на языке С включать команды на языке ассемблера. Например, инструкция

```
__asm__("Move  #0x40,%PSR");
```

дает указание компилятору С вставить в откомпилированный код ассемблерную команду

```
Move  #$40,PSR
```

которая загружает в регистр PSR значение $40.

Несколько слов о программе обработки прерываний. Она должна быть написана в виде функции на языке С. Однако компилятор реализует функции как подпрограммы.

```
#define RBU (volatile char *) 0xFFFFFFE0
#define PAOUT (char *) 0x0FFFFFFF1


void main()
{


}


void intserv()
{
  *PAOUT = *RBUF;    /*Пересылка символа в порт А*/
}
```

Рис. 9.15. Вызов функции в программе на языке С

На рис. 9.15 дан пример реализации функции на языке С. Здесь же показана главная программа, выполняющая некоторую задачу. Ее инструкций мы не приводили. Кроме того, на этом рисунке представлена функция intserv, которая просто пересылает один символ из буфера приема RBUF в выходной порт PAOUT. Сгенерированный компилятором код данной функции состоит из двух команд:

```
Move     $FFFFFFE0,$FFFFFFF1
ReturnS
```

Как же сделать функцию intserv обработчиком прерываний? Для возвращения из программы обработки прерываний должна применяться специальная команда ReturnI. Эта команда приводит к восстановлению счетчика команд и регистров

состояния процессора к тому состоянию, в котором они находились до прерывания. Таким образом, нам нужно включить в программу команду ReturnI, для чего используется следующая инструкция:

```
__asm__("ReturnI");
```

В результате откомпилированный код будет таким:

```
Move    $FFFFFFE0,$FFFFFFF1
ReturnI
ReturnS
```

Конечно, команда ReturnS при таком условии никогда не будет выполнена.

Теперь можно написать программу для нашего примера. Один из ее возможных вариантов в стиле программы с рис. 9.11 приведен на рис. 9.16.

```
/*Определение адресов регистров*/
#define RBUF (volatile char *) 0xFFFFFFE0
#define SCOUNT (char *) 0xFFFFFFE3
#define PAUOT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define int_addr (int *) (0x24)

void intserv();

void main()
{
  /*Инициализация параллельного порта*/
  *PADIR = 0xFF;              /*Конфигурировние порта A как выходного*/

  /*Инициализация механизма прерываний*/
  int_addr = &intserv;        /*Задание вектора прерывания*/
  __as__("Move #0x40,%PSR");/*Процессор отвечает на прерывания IRQ*/
  *SCONT = 0x10;              /*Разрешение на прерывание от приемника*/

/*Пересылка символов*/
  while (1);                  /*Бесконечный цикл*/
}

/*Программа обработки прерываний*/
void intserv()
{
  *PAOUT = *RBUF;             /*Пересылка символа в порт A*/
  __asm__("ReturnI");         /*Возврат из прерывания*/
}
```

Рис. 9.16. Программа на языке C, выполняющая пересылку символов с использованием прерываний

Обратите внимание, что указатели на регистры ввода-вывода имеют символьный тип, поскольку указывают на данные длиной в 1 байт. А вот указатель int_addr имеет тип беззнакового целого, так как указывает на 4-байтовый вектор прерывания.

9.5. Временные ограничения устройств ввода-вывода

Приведенный в предыдущем разделе пример получился таким простым потому, что подключенное к порту A выходное устройство работает быстрее устройства, передающего символы через последовательный интерфейс. Однако на практике используются устройства с очень разной скоростью функционирования. Предположим, что в нашем примере выходное устройство работает медленнее входного. Это означает, что символы не могут непосредственно пересылаться из RBUF в PAOUT — их приходится временно сохранять в буфере памяти, который можно организовать, скажем, в виде очереди (FIFO). Однако у данной структуры, очереди, имеется один серьезный недостаток: указатели ее начала и конца все время увеличиваются, из-за чего она по мере записи и чтения символов смещается в памяти. Поэтому удобнее использовать циклический буфер или циклическую очередь, для которой выделяется фиксированная область памяти. Когда циклическая очередь достигает конца выделенной для нее области памяти, она перемещается в начало этой области. Конечно, такой буфер может переполниться, если более быстрое устройство сгенерирует слишком много символов и входное устройство не успеет их принять. Для того чтобы избежать проблем, связанных с переполнением, в нашем примере передающее устройство генерирует символы пакетами, длина которых не может превышать 80 символов, причем принимающее устройство обрабатывает их до прибытия следующего пакета. Это означает, что циклический буфер должен вмещать не более 80 символов.

Циклический буфер можно реализовать в виде массива 8-разрядных элементов и двух индексов, указывающих на начало и конец очереди. Когда эти индексы равны, очередь пуста.

9.5.1. Программа на языке C для пересылки символов с использованием циклического буфера

На рис. 9.17 приведен пример программы на языке C, в которой пересылка символов осуществляется через циклический буфер. Роль такого буфера играет массив mbuffer, состоящий из 80 элементов. Символы помещаются в буфер по индексу fin, а извлекаются из него по индексу fout. На каждой итерации цикла проверяется регистр состояния SSTAT, что позволяет узнать, есть ли в RBUF новый символ. Если в RBUF нет нового символа, то при условии, что циклический буфер не пуст и порт A готов принять данные, выполняется пересылка таковых. После каждой операции пересылки значения индексов обновляются.

```
/*Определение адресов регистров */
#define RBUF (volatile char *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#define PAUOT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PSTAT (volatile char *) 0xFFFFFFF6
#define BSIZE 80

void main()
{
  unsigned char mbuffer[BSIZE];
  unsigned char fin, fout;
  unsigned char temp;

  /*Инициализация порта А и циклического буфера*/
  PADIR = 0xFF;                    /*Конфигурирование порта А как выходного*/
  fin = 0;
  fout = 0;

  /*Пересылка символов*/
  while (1){                       /*Бесконечный цикл*/
    while ((SSTAT & 0x1) == 0) {   /*Ожидание нового символа*/
      if (fin != fout)             /*Если циклический буфер не пуст*/
        if (*PSTAT & 0x2) {        /*и выходное устройство готово,*/
          *PAOUT = mbuffer[fout]; /*происходит пересылка символа в порт А*/
          if (fout < BSIZE-1)      /*Обновление индекса конца очереди*/
            fout++;
          else
            fout = 0;
        }
      }
    }
    mbuffer[fin]=*RBUF;            /*Считывание символа из буфера приема*/
      if (fin < BSIZE-1)           /*Обновление индекса начала очереди*/
        fin++;
      else
        fin = 0;
  }
}
```

Рис. 9.17. Программа на языке С, выполняющая пересылку символов через циклический буфер

9.5.2. Программа на языке ассемблера для пересылки символов с использованием циклического буфера

На рис. 9.18 показано, как должна выглядеть аналогичная программа на языке ассемблера. Для доступа к циклическому буферу здесь используется индексный режим адресации. Регистр R0 указывает на начало буфера, а регистры R1 и R2

содержат индексы начала и конца очереди. В остальном программа работает так же, как программа на рис. 9.17.

```
RBUF        EQU         $FFFFFFE0   Приемный буфер
SSTAT       EQU         $FFFFFFE2   Регистр состояния последовательного
                                    интерфейса
PAOUT       EQU         $FFFFFFF1   Выходные данные порта A
PADIR       EQU         $FFFFFFF2   Регистр направления порта A
PSSTAT      EQU         $FFFFFFF6   Регистр состояния параллельного
                                    интерфейса
MBUFFER     ReserveByte 80          Определение циклического буфера
* Инициализация
            ORIGIN      $1000
            MoveByte    #$FF,PADIR  Конфигурирование порта A в качестве
                                    выходного
            Move        #MBUFFER,R0 R0 указывает на буфер
            Move        #0,R1       Инициализация указателя на начало
                                    очереди
            Move        #0,R2       Инициализация указателя на конец
                                    очереди
* Пересылка символов
LOOP        Testbit     #0,SSTAT    Проверка того, готов ли новый символ
            Branch≠0    READ
            Compare     R1,R2       Проверка того, пуста ли очередь
            Branch=0    LOOP        Очередь пуста
            Testbit     #1,PSTAT    Проверка готовности порта A
            Branch=0    LOOP
            MoveByte    (R0,R2),PAOUT Отправка символа в порт A
            Add         #1,R2       Увеличение значения указателя на
                                    конец очереди
            Compare     #80,R2      Проверка того, вышел ли указатель
                                    за границу буфера
            Branch<0    LOOP
            Move        #0,R2       Переустановка его на начало буфера
            Branch      LOOP
READ        MoveByte    RBUF,(R0,R1) Помещение в очередь нового символа
            Add         #1,R1       Увеличение значения указателя на
                                    начало очереди
            Compare     #80,R1      Проверка того, вышел ли указатель
                                    за границу буфера
            Branch<0    LOOP
            Move        #0,R1       Переустановка на начало буфера
            Branch      LOOP
```

Рис. 9.18. Программа на унифицированном языке ассемблера, выполняющая пересылку символов с использованием циклического буфера

9.6. Таймер реакции

Теперь вы знаете, что собой представляют базовые функции и элементы микроконтроллера, и теперь интересно было бы посмотреть, каким образом его можно задействовать в каком-нибудь простом устройстве. Реальные устройства со встроенными процессорами достаточно сложны, но мы воспользуемся более простым примером — создадим так называемый таймер реакции, с помощью которого можно измерять скорость реагирования человека на визуальные стимуляторы. Наш микроконтроллер должен включать свет, а затем определять время, которое потребуется человеку на то, чтобы выключить свет нажатием кнопки.

Вот подробное описание таймера.

- Система активизируется нажатием кнопки «Пуск».
- После активизации на дисплее (состоит из трех 7-сегментных индикаторов) выводится 000. Светодиод не горит.
- Через 3 с светодиод зажигается и начинается отсчет времени.
- После нажатия кнопки «Стоп» отсчет времени прекращается и светодиод гаснет, а на дисплее выводится истекшее время.
- Время вычисляется и выводится в сотых долях секунды. Поскольку на дисплее всего лишь три цифры, предполагается, что истекшее время составляет менее 10 с.

Аппаратная схема нашего таймера приведена на рис. 9.19. Все элементы прибора, за исключением кнопок, дисплея и памяти, входят в состав микроконтроллера.

Для всех операций ввода-вывода мы будем использовать параллельные порты A и B. Две старшие BCD-цифры на дисплее подключены к порту A, а младшая цифра — к порту B. Кнопки и светодиоды соединены с четырьмя младшими разрядами порта B, как показано на рисунке. Для измерения времени применяется счетчик/таймер, который управляется системным тактовым сигналом с частотой 100 МГц.

Программа для нашего прибора выполняет следующие действия:

- чтобы вовремя узнать о нажатии пользователем кнопки «Пуск», реализует бесконечный цикл, в котором проверяет состояние этой кнопки;
- определив, что кнопка «Пуск», нажата, то есть $PB_1 = 0$, после задержки в 3 с зажигает светодиод;
- присваивает счетчику начальное значение $FFFFFFF1 и активизирует процесс отсчета (значение счетчика увеличивается с каждым тактовым импульсом);
- опрашивает в цикле состояние кнопки «Стоп», с тем чтобы зафиксировать момент ее нажатия пользователем;
- обнаружив, что кнопка нажата, останавливает счетчик и вычисляет истекшее время;
- преобразует полученное значение в формат BCD и направляет его на дисплей.

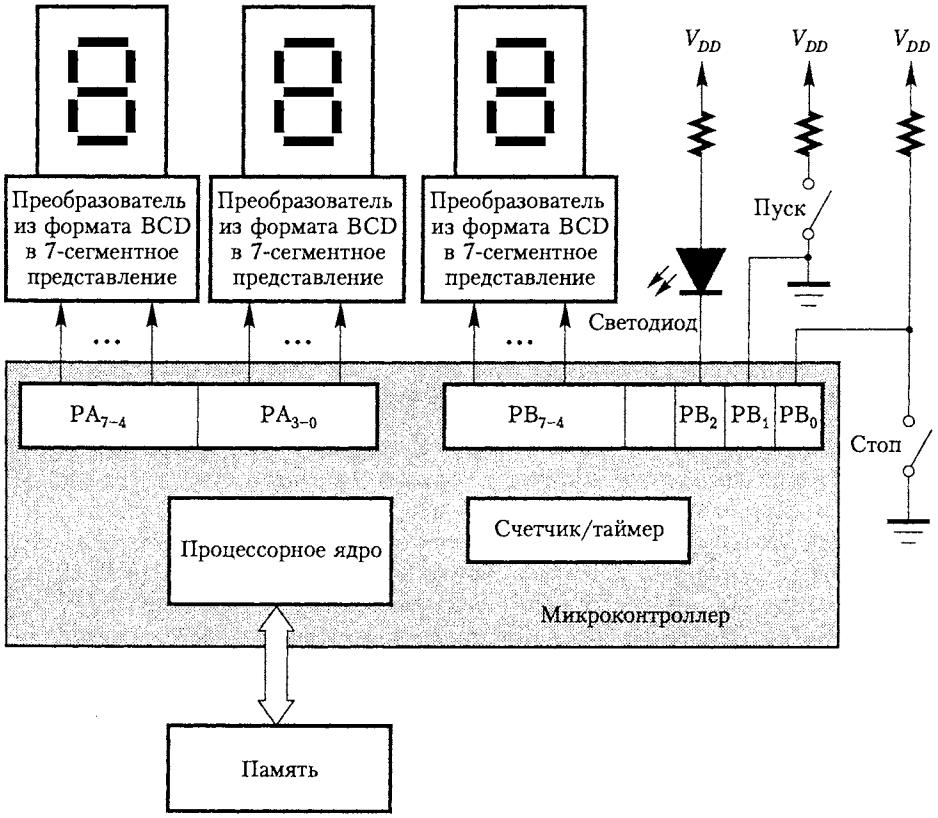


Рис. 9.19. Схема таймера реакции

Адреса регистров ввода-вывода микроконтроллера приведены на рис. 9.6–9.9. Программа должна сконфигурировать порты A и B, как показано на рис. 9.19. Все разряды порта A и четыре старших разряда порта B конфигурируются как выходные. Из четырех младших разрядов порта B два, PB_0 и PB_1, используются как входные, а один, PB_2, — как выходной. Управляющие сигналы двух портов не применяются, поскольку входное устройство состоит из простых ключей (кнопок), а выходным устройством является дисплей, который немедленно отражает изменения сигналов на управляемых им контактах порта.

Мы покажем, как запрограммировать наше устройство на языке C и на языке ассемблера.

В обеих программах имеются команды, выполняющие следующие задачи. После нажатия кнопки «Пуск» с помощью таймера организуется 3-секундная задержка. Поскольку счетчик/таймер тактируется сигналом с частотой 100 МГц, в счетчик записывается шестнадцатеричное значение 11E1A300, соответствующее десятичному значению 300000000. Процесс обратного отсчета начинается с установки разряда $CTCONT_0$ в 1. Когда счетчик достигает значения 0, включается светодиод и начинается отсчет времени, для чего счетчик устанавливается в FFFFFFFF. Обнаружив, что нажата кнопка «Стоп», программа останавливает

процесс отсчета, для чего она устанавливает $CTCONT_1$ в 1. Общее количество отсчитанных тактов вычисляется так:

Отсчитано тактов = 0xFFFFFFFF − Значение счетчика

а реальное время в сотых долях секунды — так:

Реальное время = (Отсчитано тактов)/1000000

Это двоичное целочисленное значение можно преобразовать в десятичный формат, разделив его на 100 для получения старшей цифры, потом разделив остаток на 10 для получения следующей цифры, а последний остаток будет представлять собой младшую цифру.

9.6.1. Программа для таймера реакции на языке C

На рис. 9.20 приведена программа на языке C, управляющая нашим устройством. Сначала эта программа конфигурирует порты A и B, затем выключает дисплей и светодиод, а затем начинает опрос, чтобы определить значение сигнала на контакте PB_1. После нажатия кнопки «Пуск» значение разряда PB_1 становится равным 1 и происходит 3-секундная задержка. Потом включается светодиод и начинается отсчет времени реакции. В ожидании нажатия кнопки «Стоп» выполняется второй цикл. Когда эта кнопка будет нажата, светодиод выключается, счетчик останавливается и считывается его содержимое. Как вычисляется истекшее время и как полученное значение преобразуется в десятичное число, описано выше. Результирующие три цифры помещаются в регистры данных портов (рис. 9.19).

```
/*Определение адресов регистров*/
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBIN (volatile char *) 0xFFFFFFF3
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5
#define CNTM (int *) 0xFFFFFFD0
#define COUNT (volatine int *) 0xFFFFFFD4
#define CTCON (char *) 0xFFFFFFD8

void main()
{
  unsigned int cuonter_value, totel_count;
  unsigned int actual_time, seconds, tenths, hundredths;

/*Инициализация параллельных портов*/
  *PADIR = 0xFF;              /*Конфигурирование порта A*/
  *PBDIR = 0xF4;              /*Конфигурирование порта B*/
  *PAOUT = 0x0;               /*Выключение дисплея*/
  *PBOUT = 0x4;               /*и светодиода*/
```

Рис. 9.20. Программа для таймера реакции на языке C

```
    /*Начало теста*/
    while (1) {                          /*Бесконечный цикл*/
      while ((*PBIN & 0x2) == 0);        /*Ожидание нажатия кнопки «Пуск»*/

      /*Пауза в 3 с и включение светодиода*/
      *CNTM = 0x11E1A300;     /*Установка для таймера значения 300000000*/
      *CNCOUNT = 0x1;                    /*Запуск таймера*/
      while ((*CNSTAT & 0x1) == 0); /*Ожидание установки таймера в нуль*/
      *PBOUT                             /*Включение светодиода*/

      /*Инициализация процесса счета*/
      counter_value = 0;
      *CNTM = 0xFFFFFFFF;  /*Установка начального значения счетчика*/
      *CTCONT = 0x1;                     /*Начало отсчета*/

      while ((*PBIN & 0x1) == 0);        /*Ожидание нажатия кнопки «Стоп»*/

      /*Кнопка «Стоп» нажата — остановка начала отсчета*/
      *PBOUT = 0x4;                      /*Выключение светодиода*/
      *CTCONT = 0x2;                     /*Остановка счетчика*/
      counter_value = *COUNT;            /*Считывание содержимого счетчика*/

      /*Вычисление общего количества тактов*/
      total_count = (0xFFFFFFFF - counter_value);

      /*Преобразование количества тактов в значение времени*/
      actual_time = total_count / 1000000     /*Время в сотых долях секунды*/
      seconds = actual_time / 100;
      tenths = (actual_time - seconds * 100) / 10;
      hundredths = actual_time - (seconds * 100 + tenths * 10);

      /*Вывод прошедшего времени*/
      *PAOUT = ((seconds < 4 | tenths);
      *PBOUT = ((hundredths < 4) | 0x4);     /*Светодиод выключен*/
    }
}
```

Рис. 9.20 (*продолжение*)

9.6.2. Программа для таймера реакции на языке ассемблера

На рис. 9.21 показано, как реализовать описанную выше программу на языке ассемблера. В основу такой программы будет положена общая стратегия, описанная в разделе 9.6. Комментарии к каждой команде помогут читателю понять, как она работает.

Для преобразования в формат BCD используется команда деления, описанная в разделе 2.10.3. Данная команда делит содержимое регистра R2 на содержимое регистра R1. Результат операции помещается в регистр R2, а остаток от деления

помещается в регистр R3. Обратите внимание, что после деления истекшего времени (выраженного в сотых долях секунды) на 100 нам нужны только четыре младших разряда частного, находящегося в регистре R2. Предполагаем, что истекшее время будет меньше 10 с, так что для его отображения достаточно трех цифр. Первая цифра делителя помещается в R4. Остаток перемещается из R3 в R2, и выполняется деление на 10.

```
PAOUT   EQU     $FFFFFFF1       Выходные данные порта A
PADIR   EQU     $FFFFFFF2       Регистр направления порта A
PBIN    EQU     $FFFFFFF3       Входные контакты порта B
PBOUT   EQU     $FFFFFFF4       Выходные данные порта B
PBDIR   EQU     $FFFFFFF5       Регистр направления порта B
CNTM    EQU     $FFFFFFD0       Начальное значение счетчика
COUNT   EQU     $FFFFFFD4       Содержимое счетчика
CTCONT  EQU     $FFFFFFD8       Регистр управления
* Инициализация
        ORIGIN  $1000
        MoveByte #$FF,PADIR     Конфигурирование порта A
        MoveByte #$F4,PBDIR     Конфигурирование порта B
START   MoveByte #0,PAOUT       Выключение дисплея
        MoveByte #4,PBOUT       Выключение светодиода, для чего
                                задается PB2 = 1
* Ожидание нажатия кнопки «Пуск»
GKEY    Testbit  #1,PBIN        Кнопка «Пуск» соединена
        Branch=0 GKEY           с контактом PB1
* Пауза в 3 с и включение светодиода
        Move     #11E1A300,CNTM Установка для таймера значения
                                300 000 000
        MoveByte #1,CTCONT      Запуск таймера
DELAY   Testbit  #0,CTCONT      Ожидание того, когда значение
        Branch=0 DELAY          таймера достигнет 0
        MoveByte #0,PBOUT       Включение светодиода
* Инициализация процесса отсчета
        Move     #$FFFFFFFF,CNTM Устанавливаем начальное значение
                                счетчика
        MoveByte #1,CTCONT      Начинаем отсчет
* Ожидание нажатия кнопки «Стоп»
SKEY    Testbit  #0,PB          Кнопка «Стоп» соединена
        Branch=0 SKEY           с контактом PB0
```

Рис. 9.21. Программа для таймера реакции на языке ассемблера

```
* Остановка отчета и чтение последнего значения
        MoveByte    #4,PBOUT        Выключение светодиода, для чего
                                    задается PB2 = 1
        MoveByte    #2,CTCONT       Остановка счетчика
        Move        COUNT,R0        Считывание содержимого счетчика

* Вычисление общего количества тактов
        Move        #$FFFFFFFF,R2   Определение фактического
        Subtract    R0,R2           количества тактов

* Преобразование количества тактов в значение времени, а затем в формат BCD

* Помещение двух старших цифр BCD в R4, а младшей цифры в R3
        Move        #1000000,R1     Вычисление времени
        Divide      R1,R2           в сотых долях секунды
        Move        #100,R1         Деление полученного значения на
                                    100, чтобы получить
        Divide      R1,R2           количество секунд
        Move        R2,R4           Сохранение этой цифры в R4
        Move        R3,R2           Использование остатка как
                                    следующего делимого
        Move        #10,R1          Деление его на 10, чтобы получить
                                    количество
        Divide      R1,R2           десятых долей секунды
        LShiftL     #4,R4           Помещение первых двух
        Or          R2,R4           цифр BCD в R4

* Вывод истекшего времени
        MoveByte    R4,PAOUT        Вывод первых двух цифр в порт A
        LShiftL     #4,R3           Вывод третьей цифры в порт B
        Or          #4,R3           и сохранение светодиода
        MoveByte    R3,PBOUT        в выключенном состоянии
        Branch      START           Готовность к следующему тесту
```

Рис. 9.21 (*продолжение*)

Цифры BCD, соответствующие количеству секунд и количеству десятых долей секунды истекшего времени, помещаются в младший байт регистра R4 для отправки в порт A. Третья цифра, соответствующая количеству сотых долей секунды, сдвигается в регистре R3 в разряды $R3_{7-4}$. Поскольку содержимое регистра R3 направляется в порт B, то для того чтобы сохранить светодиод включенным, необходимо установить разряд $R3_3$ в 1.

В отличие от примеров из предыдущих глав, демонстрирующих работу или структуру отдельных элементов компьютерной системы, таймер реакции представляет собой пример действующей системы с компьютерным управлением.

Важной особенностью программного обеспечения для встроенных систем является его тесное взаимодействие с аппаратным обеспечением. Системы, подобные нашему таймеру, называют *реактивными*, поскольку составляющие их программы вызываются в ответ на внешние по отношению к процессору события, такие как нажатие кнопки или поступление символа во входной порт. Как рассказывалось в главе 4, для координирования такого взаимодействия существует два механизма — опрос и прерывания. Какому из них отдать предпочтение, решает сам разработчик программного обеспечения.

9.7. Семейства встраиваемых процессоров

Итак, мы рассмотрели примеры встраиваемого процессора (раздел 9.3) и простого прибора с компьютерным управлением (раздел 9.6). Ниже коротко описываются современные серийно выпускаемые процессорные микросхемы для встраиваемых систем. Во многих таких системах мощные процессоры не нужны. Так, для микроволновой печи, речь о которой шла в разделе 9.1.1, безусловно, достаточно простейшего контроллера, поскольку в ней производятся лишь несложные вычислительные операции. В подобных случаях применяются микросхемы с простым процессором, но с количеством памяти, достаточным для того, чтобы для управления прибором использовалась единственная микросхема. А вот в описанном в разделе 9.1.2 цифровом фотоаппарате производятся гораздо более сложные вычисления, а следовательно, и процессор должен быть более мощным.

Важной характеристикой процессора является количество битов данных, параллельно обрабатываемых при обращении к памяти. Самые мощные микроконтроллеры содержат 32-разрядный процессор с 32-разрядной шиной данных. Такие характеристики имеют некоторые микроконтроллеры на основе архитектуры ARM. Кроме того, существуют процессоры с внутренней 32-разрядной структурой и 16-разрядной шиной, соединяющей процессор с памятью. Примером может служить семейство микроконтроллеров 683xx компании Motorola, в которых используется процессорное ядро 68000. Такие микроконтроллеры классифицируются как 16-разрядные. Но наиболее популярными стали 8-разрядные микроконтроллеры, которые гораздо дешевле, хотя и вполне удовлетворяют нуждам большинства встраиваемых систем. Существуют и 4-разрядые микросхемы, популярные благодаря своей простоте и очень малой стоимости.

9.7.1. Микроконтроллеры на основе процессоров Intel 8051

В начале 1980-х годов корпорация Intel выпустила микросхему микроконтроллера 8051. Она основывалась на архитектуре семейства процессоров Intel 8080, которые представляли собой 8-разрядные микросхемы универсального назначения. Микросхема 8051 быстро завоевала популярность и стала одной из самых распространенных микросхем такого класса. У нее было четыре 8-разрядных порта ввода-вывода, интерфейс UART и два 16-разряных счетчика/таймера. Кроме того, она содержала 4 Кбайт ROM и 128 Кбайт RAM. Эта же микросхема выпускалась в EPROM-версии под кодом 8751, и в ней вместо ROM было 4 Кбайт памяти EPROM.

Микроконтроллерные микросхемы могут производиться как по технологии n-МОП, так и по технологии КМОП. КМОП-микросхемы потребляют меньше энергии, поэтому особенно привлекательны для устройств, работающих от батареек. КМОП-версии описанных выше микросхем обозначаются как 80C51 и 80C52.

Архитектура 8051 была разработана корпорацией Intel, но со временем идентичные или расширенные микросхемы стали изготавливать и другие производители полупроводниковых устройств. С точки зрения конструкторов встраиваемых систем, выпуск альтернативных модификаций микросхем следует рассматривать как очень удачное решение, поскольку таким образом дается гарантия, что они всегда будут представлены на рынке и что их цена вследствие конкуренции начнет постепенно снижаться.

9.7.2. Микроконтроллеры компании Motorola

Начиная с 1980-х годов ведущими производителями микропроцессорных микросхем являются компании Intel и Motorola. В основе их микроконтроллеров лежат 8-разрядные микропроцессоры, пользующиеся у потребителей наибольшей популярностью. В настоящее время Motorola выпускает широчайший диапазон микропроцессоров на основе разных процессорных ядер.

Микроконтроллер 68HC11

Самыми популярными из числа изготавливаемых компанией Motorola являются микропроцессоры 6800 и 6809. Еще одна микросхема, система команд которой является надмножеством команд процессора 6800, получила название 68HC11. У нее пять портов ввода-вывода, которые могут использоваться для самых разнообразных целей. В состав подсистемы ввода-вывода входят два последовательных интерфейса, асинхронный и синхронный. Асинхронный интерфейс работает на основе стартстопного протокола, описанного в разделе 10.3.1. Синхронный интерфейс реализует схему под названием SPI (Serial Peripheral Interface Protocol — протокол последовательного периферийного интерфейса). К микроконтроллеру 68HC11 можно подключить до восьми аналоговых входных устройств, поскольку эта микросхема в состоянии выполнять аналого-цифровое преобразование. Кроме того, в ней имеется схема счетчика/таймера, способная работать в нескольких разных режимах.

Количество памяти в контроллере 68HC11 зависит от конкретной его модели. Самые «скромные» из них содержат 8 Кбайт ROM, 512 байт EEPROM и 256 байт RAM, а более мощные — 12 Кбайт ROM, 512 байт EEPROM и 512 байт RAM.

Микроконтроллеры 683xx

Семейство микроконтроллеров 683xx основано на процессорном ядре 68000. Эта микросхема включает параллельные и последовательные порты, счетчик/таймер и аналого-цифровой преобразователь. Количество памяти на микросхеме зависит от конкретной модели. Например, микросхема 68376 содержит 8 Кбайт EEPROM и 4 Кбайт RAM.

Микроконтроллеры ColdFire

В микроконтроллерах MCF5xxx, известных как встраиваемые процессоры ColdFire, используется система команд архитектуры 68000. Отличительной особенностью данных микроконтроллеров является конвейерная структура, обеспечивающая повышенную производительность. Они содержат полную 32-разрядную шину и предназначены для использования в роли «систем на одной микросхеме», о которых рассказывалось в разделе 9.9.

Микроконтроллеры PowerPC

Самые мощные микропроцессоры компании Motorola, известные под названием PowerPC, основываются на архитектуре RISC. Эта же архитектура положена и в основу микроконтроллеров семейства MCF5xxx.

9.7.3. Микроконтроллеры ARM

Архитектура ARM, описанная в главе 3, привлекательна в первую очередь для встраиваемых систем, которые должны обладать достаточной вычислительной мощностью при относительно низком энергопотреблении и невысокой цене. Она ориентирована прежде всего на разработку систем на одной микросхеме. Кроме того, микроконтроллеры ARM выпускаются в виде отдельных микросхем.

Разработана целая серия процессорных ядер ARM, предназначенных для встраиваемых систем, в том числе ARM6, ARM7, ARM9 и ARM10. Все команды базовой архитектуры ARM имеют длину 32 разряда. В еще одной версии этой архитектуры, под названием Thumb, используются 16-разрядные команды и 16-разрядная шина данных. В версии Thumb применяется и подмножество команд ARM, перекодированных для 16-разрядного формата. Кроме того, в ней меньшее по сравнению с архитектурой ARM количество регистров. Достоинством процессора Thumb является то, что для хранения его программ требуется меньший объем памяти. Во время выполнения каждая команда расширяется до нормальной 32-разрядной команды ARM. Для этого в ядре ARM, поддерживающем архитектуру Thumb, в дополнение к обычным схемам имеется *декомпрессор Thumb*.

Архитектура ARM и основанные на ней процессорные ядра разработаны компанией Advanced RISC Machines Ltd, но лицензии на их выпуск имеются и у ряда других компаний. Некоторые из них, в частности Atmel Corp., Sharp Electronics Corp., Samsung Semiconductor Inc., выпускают микроконтроллеры на основе ядра ARM (например, в микроконтроллере AT91F40416 компании Atmel используется Thumb-ядро ARM7-TDMI). Этот микроконтроллер содержит 4 Кбайт RAM, 256 Кбайт флэш-памяти, доступной только для чтения, 32 программируемые линии ввода-вывода, два последовательных порта и счетчик/таймер.

9.8. Особенности разработки микроконтроллеров

Конструкторы встраиваемых систем должны принимать множество важных решений. Устройство, для которого разрабатывается конкретная система, налагает множество требований и ограничений. В этом разделе рассматриваются самые важные из встающих перед конструкторами вопросов.

Стоимость

Электронные компоненты большинства встраиваемых систем не должны быть дорогостоящими. Самая дешевая реализация возможна при условии, что все функции системы выполняются одной микросхемой. Такая микросхема должна иметь разветвленные средства ввода-вывода и достаточный объем встроенной памяти для хранения всего необходимого программного обеспечения.

Средства ввода-вывода

Микросхемы микроконтроллеров обычно содержат множество разных ресурсов ввода-вывода. В простейших из них имеются только параллельные и последовательные порты, а более сложные включают также счетчик, таймер, схемы ЦАП и АЦП.

Важной характеристикой микросхемы является количество линий ввода-вывода. Когда их недостаточно, приходится использовать дополнительные внешние схемы, как в примере с таймером реакции на рис. 9.19. В этом примере микроконтроллер выдавал 4-разрядные коды BCD, которые преобразовывались внешней схемой декодирования в изображения цифр на дисплее. Но если бы вместо двух параллельных портов этот микроконтроллер имел четыре, каждый 7-сегментный индикатор можно было бы подключить к отдельному порту, а управляющая программа генерировала бы выходные сигналы для управления каждым из 7 сегментов индикатора.

Разрядность

Микроконтроллерные микросхемы могут иметь разную разрядность. Если системой способен управлять 8-разрядный микроконтроллер, нет смысла использовать 16-разрядную схему, которая значительно дороже, больше по размеру и потребляет большее количество энергии. Преобладающая часть реальных систем может управляться микросхемами относительно небольшой разрядности. Статистика показывает, что наиболее востребованными в последние годы являются 8-разряные микросхемы, за ними следуют 4- и 16-разрядные.

Физический размер микросхемы важен с точки зрения места, занимаемого ею на печатной плате. Он достаточно сильно влияет на ее стоимость.

Потребляемая мощность

Важной характеристикой всех компьютерных систем считается потребляемая ими мощность. В высокопроизводительных системах она может быть очень высокой, поэтому необходим какой-нибудь механизм отвода выделяемого тепла. Что касается встроенных систем, то потребляемая ими мощность, напротив, обычно низка, и в отводе тепла нет необходимости. Однако такие системы часто питаются от батареек, поэтому их мощность — все равно очень важный показатель.

Снизить данный показатель можно путем создания микросхем с использованием КМОП-технологии — в этом случае потребляемая мощность будет пропорциональна тактовой частоте. И если для системы допустима низкая производительность, то можно снизить тактовую частоту и тем самым сократить потребляемую мощность. Еще одним компромиссным решением может стать уменьшение

функциональных возможностей микросхемы контроллера, позволяющее упростить схему и сократить количество ее элементов, что также приведет к уменьшению потребляемой мощности.

Память на микросхеме

Если микросхема микроконтроллера содержит достаточно памяти, ее можно использовать в качестве простой встроенной системы. Объем и тип памяти при этом могут быть очень разными. Для хранения данных во время вычислений достаточно сравнительно небольшого объема RAM-памяти. Программы могут храниться в памяти, доступной только для чтения, объем которой должен быть намного больше. Это может быть память типа ROM, PROM, EPROM, EEPROM или флэш-память (стоимость возрастает в порядке перечисления типов). Если нужна память очень большого объема, дешевле всего использовать ROM, но это наименее гибкий вариант, поскольку содержимое ROM раз и навсегда заносится в микросхему при ее производстве. Память типа PROM и EPROM можно программировать в процессе производства устройства со встроенным контроллером. А наиболее гибкими являются микросхемы EEPROM и флэш-памяти, которые можно перепрограммировать по многу раз.

Производительность

Для бытовой техники и игрушек, за исключением видеоигр вроде Sony PlayStation, производительность — не самое важное требование. Поэтому для них прекрасно подходят небольшие по размеру и недорогие микросхемы. Но для таких устройств, как сотовые телефоны и портативные видеоигры, необходимы высокопроизводительные контроллеры. Большая производительность — это более мощные микросхемы, более высокая стоимость и большее количество потребляемой энергии. А поскольку такие системы, тем не менее, часто работают от батареек, важно предельно снизить потребляемую ими мощность. В главе 3 рассматривалась архитектура ARM, одной из реализаций которой является микросхема StrongARM, специально спроектированная в виде процессора с низкой потребляемой мощностью и достаточно высокой производительностью. Для использования во встроенных системах, для которых важны такие показатели, как стоимость и производительность, предназначена Thumb-версия архитектуры ARM, обсуждаемая в разделе 9.7.3.

Программное обеспечение

Языки высокого уровня имеют множество преимуществ по сравнению с ассемблером. Они способны ускорить процесс разработки программ, облегчают их дальнейшее сопровождение и модификацию. Однако в некоторых случаях имеет смысл все же пользоваться языком ассемблера. Хорошо написанная ассемблерная программа компилируется в объектный код, объем которого на 10–20 % меньше (с точки зрения занимаемой программой памяти), чем у объектного кода, полученного в результате компиляции программы на языке высокого уровня. И если встроенная система основана на микроконтроллере со встроенной памятью, важно чтобы весь необходимый код поместился в эту память, то есть чтобы не пришлось добавлять еще и внешние микросхемы памяти.

Конструктор ни в коем случае не должен ошибаться при оценке объема интегрированной в микросхему RAM. Эта память будет использоваться для хранения данных, в качестве временного буфера и как область стека. Вы можете написать очень компактный код, используя, скажем, язык C, но когда дело дойдет до ее применения, может оказаться, что он не помещается в имеющуюся RAM.

Система команд

Еще одним важным показателем является структура используемой процессором системы команд. Команды типа CISC позволяют писать более компактный код, чем команды типа RISC. Поэтому размер кода зависит и от типа процессора. Интересен пример Thumb-версии архитектуры ARM, в которой RISC-команды, разработанные для 32-разрядных процессоров, модифицированы в набор сжатых 16-разрядных команд. Программы для процессоров Thumb имеют на 30 % меньший объем, чем программы для полной архитектуры ARM. Напомним, что во время выполнения программы команды Thumb разворачиваются в нормальные ARM-команды (см. раздел 9.7.3).

Средства разработки

Создатели цифровых систем широко используют в своей работе разнообразные инструментальные программные средства, в том числе программные пакеты для автоматизированного проектирования (САПР), программное обеспечение операционной системы, ассемблеры, компиляторы и эмуляторы процессоров. Диапазон и доступность таких средств зависит от типа встраиваемого процессора. Очень хорошо, если процессор поддерживается сторонними производителями программного обеспечения, если для него имеются альтернативные источники программных средств и документации. И, конечно же, исключительно важно иметь хорошую документацию и получить полезные советы от производителей.

Тестирование и надежность

Печатные платы довольно трудно тестировать, особенно если они плотно заполнены микросхемами. Но архитектуру системы можно с самого начала спроектировать так, чтобы максимально упростить данный процесс. В частности, микроконтроллер может содержать схемы, способные облегчить тестирование плат, на которые он устанавливается. Например, некоторые микроконтроллеры содержат *порт доступа для тестирования* (test access port), соответствующий стандарту IEEE 1149.1, известному как Test Access Port and Boundary-Scan Architecture.

Встроенные системы должны быть устойчивыми и надежными. В частности, жизненный цикл типичного устройства должен составлять не менее пяти лет. Этим встроенные системы отличаются от персональных компьютеров, которые устаревают за гораздо более короткий срок.

9.9. Система на одной микросхеме

Встроенная система должна состоять из минимального количества микросхем — в идеале это может быть всего лишь одна микросхема. В очень простых случаях

все необходимые функции реализуются с помощью единственного коммерческого микроконтроллера. Однако в сложных системах это невозможно. Отдельные микроконтроллеры специально ориентированы на определенные задачи, которые трудно реализовать с применением универсальных микроконтроллеров. Например, микроконтроллер для видеоигр должен включать схемы обработки видеоизображений и звука. И совершенно иные требования предъявляются к микроконтроллерам для лазерных принтеров или сотовых телефонов.

Разработка сложного микроконтроллера — задача не из простых, требующая больших усилий и времени. Однако для большинства коммерческих продуктов время разработки должно быть относительно коротким. И если конструктор воспользуется существующими и достаточно удобными модулями, разработка может значительно ускориться. Одним из необходимых модулей является ядро процессора. Заключив соответствующее лицензионное соглашение, можно приобрести процессорное ядро одной из многочисленных компаний-производителей. Кроме того, можно приобрести схемы АЦП и ЦАП, а также схемы цифровой обработки сигналов (Digital Signal Processing, DSP). Собрав готовые модули, конструктор проектирует оставшиеся схемы — и система готова.

В разделе 9.7.3 мы упоминали о том, что ядро процессора ARM разработано для использования в качестве компонента более крупных систем. Еще один интересный пример представляет собой ядро CompactRISC компании National Semiconductor. Одной из его особенностей является способность масштабироваться от 8 до 64 разрядов. В нем имеется простой 3-ступенчатый конвейер и встроенная память: 40 Кбайт ROM и 1,4 Кбайт RAM. Блок шинного интерфейса добавляется только, если требуется внешняя память. Таким образом, можно подобрать ядро такой сложности, которая соответствует потребностям разрабатываемой системы.

Производители ядер и других модулей продают не микросхемы, а их структуру. По сути дела, они продают идеи, а не физические компоненты. Их продукты служат примерами *интеллектуальной собственности* и могут использоваться другими компаниями для разработки собственных микросхем.

9.9.1. Контроллеры на основе микросхем FPGA

Программируемые пользователем вентильные матрицы FPGA (Field Programmable Gate Array) представляют собой прекрасную основу для реализации систем на одной микросхеме. В отличие от контроллерных микросхем, предоставляющих конструктору набор готовых функциональных блоков, микросхемы FPGA предоставляют ему полную свободу разработки. В них легко включать определенные стандартные блоки и затем строить остальную часть системы по собственному усмотрению. Для того чтобы проиллюстрировать возможности этого подхода, мы опишем систему Excalibur от Altera Corporation.

Функциональные возможности FPGA растут с невероятной скоростью. Одна большая микросхема FPGA может содержать систему с сотнями и тысячами логических вентилей. Такие микросхемы достаточно вместительны для реализации типичных функций микроконтроллера и других схем, необходимых для создания встраиваемой системы.

Ключевым компонентом любой системы на одной микросхеме является процессорное ядро. Система Excalibur позволяет конструктору выбрать одно из двух альтернативных решений. Первое из них заключается в том, чтобы запрограммировать процессор на микросхеме с нуля, а другая — в том, чтобы использовать микросхему FPGA с готовым процессорным ядром на кристалле кремния, встроенным при ее производстве.

Программируемое процессорное ядро

В состав системы Excalibur входит программный модуль, созданный на языке описания аппаратного обеспечения Verilog и реализующий архитектуру процессора под названием Nios. Этот модуль позволяет конструктору либо выбрать для процессора один из библиотечных модулей на языке описания аппаратного обеспечения, таком как Verilog или VHDL, либо определить его самостоятельно в виде функционального блока путем интерактивного составления схемы. Конструктор может выбрать 32- либо 16-разрядную версию процессора, в зависимости от требований к производительности системы.

Модуль параллельного интерфейса, подобный приведенному на рис. 9.3.1, доступен в виде библиотечного модуля, для которого конструктор может задать параметры, соответствующие его собственным требованиям. Длина регистров выбирается в диапазоне от 1 до 32 разрядов. Конструктор может применить либо полный двунаправленный интерфейс, либо его более ограниченную версию. Например, можно задать только выходной порт, и тогда регистр выходных данных будет включен в состав схемы, а регистр входных данных нет. В результате ресурсы FPGA не станут тратиться на ненужные компоненты.

Последовательный интерфейс реализуется в форме схемы UART. Конструктор определяет необходимые параметры, такие как количество разрядов данных, количество стоповых битов и использование бита четности. Тактовая частота приема и передачи выбирается из предопределенного стандартного диапазона. Позднее ее можно будет изменить с помощью программного обеспечения контролера, если включить в схему регистр деления частоты (куда можно загружать значение, на которое следует разделить стандартную тактовую частоту). И снова конструктор может выбрать только необходимые функции, чтобы не выполнять ничего лишнего.

Модуль таймера позволяет реализовать функции таймера и счетчика, описанные в разделе 9.3.3. Его работа полностью управляется программным обеспечением контролера.

Большие микросхемы FPGA содержат много памяти. Входящие в их состав блоки памяти могут использоваться для реализации RAM и ROM встроенной системы, если, конечно, ее требования к памяти не слишком велики. Конструктор может задать объем необходимой памяти в виде количества слов и количества разрядов в слове. Если памяти на микросхеме недостаточно, можно добавить интерфейс для внешней памяти, в результате чего на контактах микросхемы FPGA будут реализованы сигналы шины памяти.

Средства автоматизированного проектирования, входящие в состав системы Excalibur, очень облегчают создание системы на основе микросхемы FPGA. В их состав входит мастер, предлагающий конструктору ввести желаемые параметры

и затем самостоятельно генерирующий нужные схемы. Таким образом, процессор и модули ввода-вывода реализуются автоматически. Они соединяются с помощью структуры, реализующей шинный протокол Nios. Следует заметить, что шина на микросхеме FPGA основана не на повторителях с тремя состояниями, как рассказывалось в главе 7. FPGA — универсальное устройство, содержащее множество логических элементов, соединенных проводниками и ключами. Повторители с тремя состояниями полезны только для конкретных целей, поэтому в типичных FPGA они не используются. Их функции можно реализовать путем использования мультиплексорных схем. Вместо одного двунаправленного соединения для каждого направления используется свой мультиплексор. И хотя для реализации этого подхода требуется большое количество вентилей, это хорошее решение, поскольку необходимые логические элементы и соединения составляют очень маленькую часть общего количества ресурсов FPGA.

Процессор и подсистема интерфейса занимают относительно небольшую часть микросхемы FPGA. Остальная ее часть предоставляется для реализации схем, специфических для конкретного устройства. Эти схемы можно либо непосредственно соединить с шиной процессора, либо подключить к портам ввода-вывода. Если вся система расположена на одной микросхеме, порты ввода-вывода в подсистеме процессора не обязательно должны быть соединены с контактами ввода-вывода микросхемы FPGA. Конструктор может их использовать для подключения к процессору специфических для данной системы схем.

Процессор Nios имеет систему команд типа RISC. Его производительность может достигать 50 MIPS (Million Instructions Per Second), то есть миллионов команд в секунду. Конструктор может реализовать на одной микросхеме FPGA даже несколько процессоров, создав таким образом мультипроцессорную систему.

Аппаратное процессорное ядро

В качестве альтернативы программируемому процессорному ядру можно использовать кремниевый процессор на специализированной микросхеме FPGA. Система Excalibur предоставляет такие FPGA на основе разных процессоров. Примером может служить FPGA с процессорным ядром ARM, реализованным в одной части устройства. В дополнение к процессорным схемам реализованы шина процессора ARM, модуль RAM и последовательный модуль UART. Подобным образом можно создать значительно более мощную систему, и при этом как обычно запрограммировать остальные необходимые ресурсы, для которых в микросхеме оставлена стандартная FPGA-часть. Система с жестким процессорным ядром может достигать производительности в сотни MIPS.

С точки зрения конструктора

Конструктор встроенной системы, безусловно, ищет простейшее и самое дешевое решение. И самым лучшим выбором может оказаться микросхема микроконтроллера, содержащая ресурсы для реализации всей этой системы. Совсем иная ситуация получается в том случае, если для реализации системы необходимы дополнительные микросхемы. Тогда привлекательным решением становится использование микросхем FPGA, позволяющих создать систему с минимумом микросхем.

Еще одним важным фактором, от которого зависит выбор решения, является наличие готовых модулей. Микросхема микроконтроллера содержит множество разных модулей, но все те необходимые модули, которых она не включает, приходится реализовывать в виде дополнительных микросхем. Технология FPGA позволяет конструктору создавать любые логические схемы. Многие системы содержат схемы для выполнения достаточно типичных задач, как, например, интерфейсы ввода-вывода или схемы таймера. Такие схемы должны реализовываться в виде библиотечных модулей. Хорошо, если имеются и другие полезные модули. Скажем, для систем обработки сигналов библиотека может содержать типичные фильтры и быстрые умножители. Или же, если разрабатываемая система должна подключаться к другому компьютеру через стандартную шину, такую как PCI, задача конструктора будет проще, если интерфейс PCI уже реализован в виде готового модуля.

9.10. Резюме

Настоящая глава представляет собой введение в тему разработки полных встроенных компьютерных систем. Поскольку изложенные в ней принципы достаточно универсальны, мы не говорили о каких-либо конкретных микроконтроллерах, а попытались сконцентрировать ваше внимание на ключевых задачах, стоящих перед конструктором встроенной системы.

Важной особенностью таких систем является тесное взаимодействие между аппаратным и программным обеспечением. Конструктор принимает множество решений, выбирая, например, между опросом устройств ввода-вывода и прерываниями, между различными системами команд, обеспечивающими разные возможности и разный объем результирующего кода, ищет компромисс между снижением потребления энергии и повышением производительности и т. д.

Упражнения

9.1. Микроконтроллер, описанный в разделе 9.3, принимает через последовательный порт десятичные цифры (от 0 до 9) в виде кодов ASCII. По прибытии каждой цифры он выводит ее на 7-сегментном индикаторе, подключенном к параллельному порту A. Покажите соединения, необходимые для выполнения этой задачи. Пометьте сегменты индикатора, как показано на рис. A.33. Напишите для данного микроконтроллера программу на языке C. Для определения того, поступил ли символ, используйте механизм опроса.

9.2. Напишите программу на языке ассемблера, выполняющую ту же задачу, что и программа на языке C в упражнении 9.1.

9.3. Выполните упражнение 9.1 еще раз, используя прерывания, сообщающие о поступлении очередного символа ASCII.

9.4. Напишите программу на языке ассемблера для упражнения 9.3.

9.5. Микроконтроллер, описанный в разделе 9.3, принимает через последовательный порт десятичные числа. Каждое число состоит из двух цифр, представленных как символы ASCII. Для разделения последовательных двузначных чисел используется символ H. Таким образом, два последовательных числа 43 и 28 будут представлены как H43H28. Каждое число должно быть выведено на двух 7-сегментных индикаторах, соединенных с параллельными портами A и B. Символ-разделитель выводиться не должен. Число на дисплее должно сменяться только после получения второй цифры нового числа. Покажите соединения, необходимые для выполнения этой задачи. Напишите для данного микроконтроллера программу на языке C. Для получения информации о поступлении очередного символа используйте механизм опроса.

9.6. Напишите программу на языке ассемблера, выполняющую ту же задачу, что и программа на языке C в упражнении 9.5.

9.7. Повторите упражнение 9.5 с использованием прерываний, сообщающих о поступлении очередного символа ASCII.

9.8. Напишите ассемблерную программу для упражнения 9.7.

9.9. Микроконтроллер, описанный в разделе 9.3, принимает через последовательный порт десятичные числа. Каждое число состоит из четырех цифр, представленных как символы ASCII. Для разделения последовательных двузначных чисел используется символ H. Таким образом, два последовательных числа 2143 и 6292 будут представлены как H2143H6292. Каждое число должно быть выведено на четырех 7-сегментных индикаторах. Предположим, что каждый индикатор имеет схему преобразования из формата BCD в 7-сегментное представление, как показано на рис. У9.1. Покажите необходимые соединения с микроконтроллером. Напишите для этого микроконтроллера программу на языке C. Для получения информации о поступлении очередного символа используйте механизм опроса.

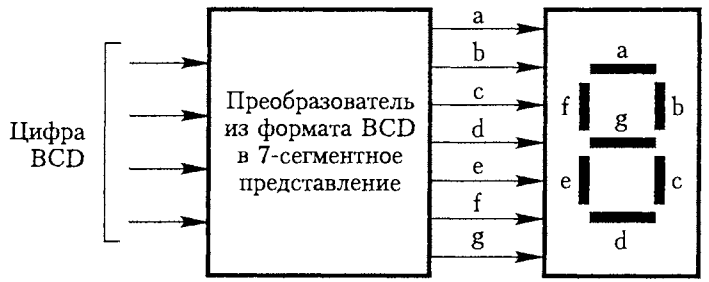


Рис. У9.1. Схема 7-сегментного индикатора с декодером BCD

9.10. Напишите программу на языке ассемблера, выполняющую ту же задачу, что и программа на языке C в упражнении 9.9.

9.11. Повторите упражнение 9.9 с использованием прерываний, которые должны сообщать о поступлении очередного символа ASCII.

9.12. Напишите программу на языке ассемблера для упражнения 9.11.

9.13. Повторите упражнение 9.9 при условии, что с 7-сегментным индикатором связан не декодер BCD, а 7-разрядный регистр. У этого регистра имеется управляющий вход Load, работающий таким образом, что семь битов данных загружаются в регистр, когда Load = 1. Каждый разряд регистра управляет одним сегментом индикатора. На рис. У9.2 показана схема соединения между регистром и индикатором. Организуйте выходные соединения микроконтроллера таким образом, чтобы через параллельный порт A выводились данные для всех четырех индикаторов.

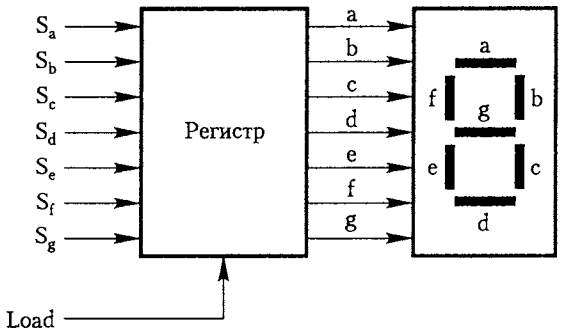


Рис. У9.2. Схема 7-сегментного индикатора с регистром

9.14. Повторите упражнение 9.13, но напишите программу на языке ассемблера.

9.15. Повторите упражнение 9.13 с использованием прерывания, сообщающего о поступлении очередного символа ASCII.

9.16. Напишите программу на языке ассемблера, выполняющую ту же задачу, что и программа на языке C в упражнении 9.15.

9.17. В разделе 9.5 мы предполагали, что передающее устройство генерирует символы пакетами, максимальная длина которых составляет 80 символов. Будут ли программы, приведенные на рис. 9.17 и 9.18, правильно работать при длине пакетов свыше 80 символов? Если нет, покажите, как их модифицировать.

9.18. В программе на рис. 9.17 для того чтобы выяснить, пуст ли циклический буфер, мы сравнивали индексы fin и fout. Вместо этого можно было бы ввести переменную-счетчик M, указывающую текущее количество символов в буфере. Модифицируйте программу для реализации этого подхода.

9.19. Повторите упражнение 9.18 для программы, представленной на рис. 9.18.

9.20. Модифицируйте таймер реакции, описанный в разделе 9.6, с учетом условия, согласно которому тестируемый человек всегда реагирует менее чем за 1 с. Для вывода времени реакции достаточно будет двух цифр, представляющих сотые доли секунды. Подключите два 7-сегментных индикатора к порту A и модифицируйте программы на рис. 9.20 и 9.21.

9.21. В схеме, представленной на рис. 9.19 в состав каждого 7-сегментного индикатора входит декодер BCD. Поэтому микроконтроллер одновременно выдает 4-разрядные коды всех выводимых на дисплее цифр. Предположим, что вместо декодера каждый индикатор будет содержать 7-разрядный регистр с управляющим входом Load, работающим таким образом, что семь битов данных загружаются в этот регистр, когда Load = 1. Каждый разряд регистра управляет одним сегментом индикатора. На рис. У9.2 показана схема соединения между регистром и индикатором. Модифицируйте программы, приведенные на рис. 9.20 и 9.21, таким образом, чтобы можно было использовать эту схему.

9.22. На рис. 9.21 двоичное число, представляющее время реакции таймера в сотых долях секунды, преобразуется в формат BCD путем последовательного деления сначала на 100, а затем на 10. Еще один способ выполнения этого преобразования заключается в последовательном делении на 10, при котором каждый остаток будет представлять очередную цифру BCD. Каким будет порядок цифр в этом случае? Модифицируйте программу на рис. 9.21 для выполнения такого преобразования.

9.23. Воспользуйтесь микроконтроллером, описанным в разделе 9.3, для создания часов реального времени. Время дня (в часах и минутах) должно выводиться на дисплее, состоящем из 7-сегментных индикаторов. Предположим, что с каждым индикатором связан декодер BCD-формата, как на рис. У9.1. Предположим также, что используется тактовая частота 100 МГц. Укажите необходимые аппаратные соединения и напишите соответствующую программу.

9.24. Повторите упражнение 9.20 при условии, что с каждым 7-сегментным индикатором связан регистр, как на рис. У9.2.

9.25. В системе процессор и память реализованы на одной микросхеме. Нужна ли для такой системы кеш-память? Поясните свой ответ.

Глава 10

Периферийные устройства

- Работа устройств ввода-вывода
- Работа сканеров и принтеров
- Графические платы и обработка графики
- Синхронные и асинхронные последовательные соединения
- Высокоскоростные соединения в сети Интернет с использованием ADSL и кабельных модемов

В предыдущих главах были рассмотрены аппаратные и программные компоненты процессоров, памяти, дисков и компакт-дисков. Мы говорили о том, как компьютер взаимодействует с внешними устройствами, обсуждали аппаратные и программные средства программно-управляемого ввода-вывода, прерываний и прямого доступа к памяти. В этой главе описывается наиболее распространенное периферийное компьютерное оборудование и рассказывается о том, как оно подключается к компьютерной системе.

Под термином *периферия* подразумевают любые подключенные к компьютеру внешние устройства. Собственно компьютер в данном случае — это только процессор и память. Все периферийные устройства можно разделить на две категории в соответствии с их назначением. Устройства первой категории выполняют операции ввода-вывода. Это клавиатура, мышь, трекбол, принтер и монитор. Ко второй категории относятся устройства, предназначенные главным образом для хранения данных, то есть вторичные или внешние запоминающие устройства (первичным запоминающим устройством является основная память компьютера). Некоторые устройства массовой памяти, в частности магнитные диски, используются в качестве статических носителей информации, позволяющих считывать и записывать данные в оперативном режиме. Другие, такие как оптические диски, гибкие диски и магнитные ленты, применяются в качестве съемных носителей — их можно вынимать из дисковода для переноса данных из одной компьютерной системы в другую. Например, с целью распространения программного обеспечения чаще всего используются оптические диски, называемые также компакт-дисками. О вторичных запоминающих устройствах подробно рассказывалось в главе 5.

В современной компьютерной периферии важное место занимают устройства, предназначенные для подключения к Интернету. Невероятно стремительное развитие компьютерной отрасли за последние годы связано со «сращиванием» компьютеров и средств коммуникации, а также с острой необходимостью в новых

средствах для World Wide Web. Результаты этого развития отражаются на всех аспектах нашей жизни.

В данной главе приводится обзор разнообразных устройств ввода-вывода, используемых в современных компьютерах, и краткое описание лежащих в их основе технологий. Те устройства, которые находятся вне корпуса компьютера, часто подключаются к нему с помощью последовательных соединений, проводных или беспроводных. Поэтому во второй части главы мы вкратце ознакомим вас с основами последовательной связи.

10.1. Устройства ввода

К числу устройств ввода относятся клавиатура и средства для перемещения указателя по экрану дисплея, то есть мышь, трекбол и джойстик. Кроме того, к данной категории принадлежат сканеры и цифровые фотоаппараты, используемые для получения изображений и ввода их в компьютер в виде цифровых данных.

10.1.1. Клавиатура

Безусловно, самым распространенным устройством ввода является клавиатура, обычно дополняемая мышью или трекболом. Наряду с видеодисплеем, выступающим в роли выходного устройства, перечисленные устройства обеспечивают непосредственное взаимодействие человека с компьютером.

Клавиатуры бывают двух типов. Клавиатура первого типа состоит из массива механических ключей, смонтированных на печатной плате. Они организованы в виде строк и столбцов и соединены с расположенным на плате микроконтроллером. Когда пользователь нажимает одну из клавиш, ключ замыкается и контроллер идентифицирует строку и столбец, определяя, какая клавиша нажата. Сгладив дребезжание ключа (см. главу 4), контроллер генерирует код клавиши и отсылает его в компьютер через последовательное соединение.

Клавиатура второго типа имеет плоскую трехслойную структуру. Ее верхний слой состоит из пластического материала, на одной поверхности которого нарисованы клавиши, а на другую нанесены проводящие соединения. Средний слой сделан из резины с отверстиями в местах расположения клавиш. А нижний слой, металлический, в местах расположения клавиш имеет выступы. Когда пользователь прикасается пальцем к изображению клавиши на верхнем слое клавиатуры, нижняя сторона этого слоя соприкасается с металлическим выступом, замыкая электрическую цепь, точно так же, как механический ключ. Клавиатуры подобного типа дешевы и надежны, а к тому же защищены от загрязнений. Особенно часто они используются в таких системах, как торговые терминалы.

10.1.2. Мышь

Изобретение мыши в 1968 году стало важным шагом в развитии новых средств взаимодействия человека и компьютера. До этого данные вводились в компьютер

преимущественно путем набора текста на клавиатуре. Появление мыши открыло простор для новых идей, а именно возможность использования графических элементов в операционной системе, таких как окна и открывающиеся меню.

Классическая мышь представляет собой удобно ложащееся в руку оператора небольшое устройство, перемещаемое по плоской поверхности. Электронная схема воспринимает это перемещение и отсылает в компьютер информацию о расстоянии, пройденном мышью в направлении осей *X* и *Y*. Для фиксации перемещения используются механические или оптические средства. Механическая мышь содержит шарик, который соприкасается с поверхностью стола и легко вращается при перемещении мыши. Вращение шарика воспринимается специальными сенсорами и используется для приращения двух счетчиков, которые отсчитывают расстояние, пройденное мышью в направлении двух осей. Кроме того, мышь снабжается двумя или тремя кнопками. Информация от кнопок и счетчиков попадает в микроконтроллер, кодируется в трехбайтовый пакет и передается в компьютер через последовательное соединение.

В оптической мыши используется светодиод (Light-Emitting Diode, LED), освещающий поверхность, на которой располагается мышь, и светочувствительные элементы, воспринимающие отражаемый от этой поверхности свет. Некоторые модели таких устройств могут работать только на специальных ковриках с сеткой из вертикальных и горизонтальных линий. При перемещении от светлого к темному участку поверхности интенсивность отраженного света изменяется, и по этим изменениям мышь определяет пройденное расстояние.

В 1999 году Microsoft представила гораздо более сложную оптическую мышь, названную IntelliMouse. Эта мышь может использоваться практически на любой поверхности. Вместо простого светового сенсора в нее встроена цифровая камера, фиксирующая изображение маленького участка поверхности под мышью и преобразующая его в цифровое представление. Камера снимает 1500 таких изображений в секунду. И если поверхность не абсолютно ровная и гладкая, в изображениях будут отражены все изменения цвета и яркости. Сравнивая последовательные изображения, встроенный процессор мыши с большой точностью измеряет пройденное ею расстояние. Для определения расстояния от одного изображения к другому процессор использует технологию обработки сигналов, называемую корреляцией. Эта сложная вычислительная задача, как мы уже сказали, должна выполняться 1500 раз в секунду, для чего необходим мощный и дешевый встроенный процессор. Такой процессор выполняет 18 млн. команд в секунду.

Со времени изобретения мыши для выполнения подобных функций было придумано множество других устройств, к числу которых относятся трекболы, джойстики и сенсорные панели.

10.1.3. Трекбол, джойстик и сенсорная панель

Мышь позволяет оператору перемещать указатель по экрану компьютера. Для выполнения аналогичных функций было разработано множество других устройств, предназначенных для разных приложений и отвечающих разным потребностям и предпочтениям пользователей.

Принцип действия *трекбола* очень похож на принцип действия механической мыши. Трекбол содержит шарик, вмонтированный в поверхность клавиатуры. Пользователь вращает этот шарик, задавая направление перемещения указателя на экране.

Джойстик — это короткая ручка на шарнире, которую можно наклонять в любом направлении. Когда информация об изменении его положения по осям *X* и *Y* попадает в компьютер, программное обеспечение перемещает указатель на экране.

Позиция ручки может фиксироваться датчиками линейного и углового положения, как показано на рис. 10.1. Выходное напряжение потенциометров *X* и *Y* передается на два аналого-цифровых преобразователя (АЦП), выходы которых определяют положение джойстика и, соответственно, направление и расстояние перемещения.

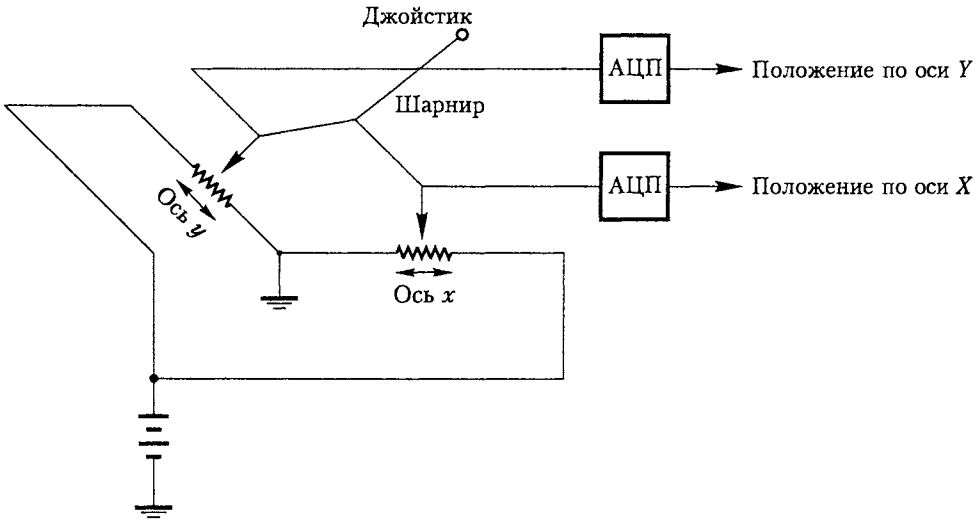


Рис. 10.1. Использование потенциометров в качестве датчиков положения джойстика

Джойстики используются в портативных компьютерах и видеоиграх. В портативном компьютере джойстик монтируется между клавишами клавиатуры и лишь слегка над ними выдается. При работе с джойстиком оператору не нужно снимать руку с клавиатуры. Перемещают его не всей рукой, как большое игровое устройство, а одним пальцем. При этом он надежен и занимает мало места. Джойстик, используемый в видеоиграх, — это отдельная достаточно большая ручка, форма которой определяется характером игры. Обычно на нем имеются кнопки, соответствующие различным командам.

Еще одним полезным устройством ввода является *сенсорная панель*. Когда пользователь касается этой панели пальцем или карандашом, электрические характеристики в точке касания изменяются. Местоположение данной точки определяется панелью и передается в компьютер. Скользя пальцем по панели, пользователь указывает, в какое место экрана нужно переместить указатель. Такая сенсорная панель является недорогим аналогом мыши или трекбола; она надежна, поскольку не

содержит движущихся частей. Чаще всего сенсорные панели применяются в портативных компьютерах.

Для использования в сенсорных панелях разработано множество новых материалов. Одной из самых передовых разработок является панель, содержащая множество тонких оптических волокон. Она может идентифицировать не только местоположение указанной пользователем точки, но и величину прилагаемого им усилия. Этот материал был создан для изготовления деталей космических роботов, но вскоре нашел множество других применений. В частности, он используется для изготовления фортепьянной клавиатуры.

Сенсорную панель можно объединить с жидкокристаллическим дисплеем. В результате получается сенсорный экран, который может служить как для ввода, так и для вывода информации. Подобные экраны обычно используются в карманных компьютерах типа PDA (Personal Digital Assistant), таких как Palm Pilot. В еще одной разновидности сенсорных экранов используется электронно-лучевая трубка. Прикосновение пальца к такому экрану вызывает изменение его емкостного сопротивления. Экраны этого типа применяются в кассовых аппаратах и торговых терминалах.

10.1.4. Сканеры

Сканер формирует цифровое представление изображения, нанесенного на лист бумаги. В первых сканерах бумага закреплялась на стеклянном цилиндре, который вращался вокруг сенсоров. В большинстве современных сканеров страница помещается на плоскую стеклянную поверхность. Она сканируется источником света, и отраженный свет фокусируется на линейную матрицу приборов с зарядовой связью (ПЗС). Когда на ПЗС попадает свет, заряжается связанный с ним маленький конденсатор, причем заряд конденсатора пропорционален силе света. С помощью аналого-цифрового преобразователя этот заряд преобразовывается в цифровую форму. В цветных сканерах используются красные, зеленые и синие фильтры, с помощью которых цвета разделяются и затем обрабатываются по отдельности. При движении источника света вдоль страницы сигнал с сенсорной матрицы считывается много раз подряд и таким образом последовательно сканируются строки пикселов изображения. Эта же технология используется и в цифровых копировальных аппаратах. По сути дела, цифровой копировальный аппарат представляет собой сканер и лазерный принтер, объединенные в одном устройстве.

После сканирования печатной страницы и пересылки ее в компьютер изображение сохраняется в памяти в виде массива пикселов. В простейшем формате каждый пиксел представляется одним битом, указывающим, является соответствующая точка темной или светлой. В изображениях более высокого качества для каждого пиксела хранится больше информации, представляющей его цвет и яркость. Ее максимальная длина составляет три байта — по одному для каждого из трех основных цветов.

Отсканированый текст в большинстве случаев представляют не в графическом, а в текстовом формате. Темные области изображения, полученного в результате сканирования текста, соответствуют напечатанным символам. Существуют технологии, позволяющие проанализировать пикселы таких областей,

сравнить их с имеющимися в памяти шаблонами символов и таким образом распознать текст и сохранить его в текстовом файле. Результирующий файл можно обрабатывать в любом текстовом процессоре, например в Word.

10.2. Устройства вывода

Выходные данные компьютера могут быть представлены в самой разной форме — в виде текста, графики, звука и т. д. Ниже описываются наиболее распространенные устройства вывода информации.

10.2.1. Дисплеи

Дисплеи служат для визуального представления выходных данных компьютера. В наиболее распространенных дисплеях используются электронно-лучевые трубки (ЭЛТ).

Давайте рассмотрим процесс формирования изображения на экране ЭЛТ. Сфокусированный электронный пучок (луч) ударяется о флюоресцентный экран и вызывает эмиссию света в виде яркой точки на темном фоне. Эта точка исчезает, когда электронный луч выключается или перемещается к другой точке. Для формирования каждой точки должны быть заданы три переменные, представляющие позицию и интенсивность луча. Его позиция задается координатами точки на экране *X* и *Y*, а интенсивность — значениями по оси *Z*, соответствующими *уровням серого цвета*. Мельчайшая адресуемая точка на экране называется *пикселом*. Она состоит из еще меньших точек разного размера, организованных в виде некоторой геометрической формы. Разные степени яркости этих точек достигаются их освещением в разных комбинациях. Данная технология называется *формированием полутонов*. На цветном дисплее каждый пиксел состоит из флуоресцентных точек трех цветов: красного, зеленого и синего. Разные цвета получаются подсветкой этих точек в разных цветовых комбинациях.

Размер точки, формируемой на экране электронным лучом, зависит от разрешающей способности дисплея. Обычно изображение на экране имеет размер от 700 до 2500 точек по осям *X* и *Y*. Информация о цвете (значения по оси *Z*) хранится в 24 битах — по одному байту для каждого цвета. Считается, что это максимальное цветовое разрешение, воспринимаемое человеческим глазом. Самым распространенным стандартом для компьютерных видеодисплеев является VGA (Video Graphics Array — логическая матрица видеографики). Стандартный дисплей VGA имеет разрешение 640×480 пикселов. Разновидности этого стандарта определяют дисплеи с более высоким разрешением, в частности 1024×768 (XVGA) и 1600×1200 (UXGA).

И текстовые, и графические изображения формируются с помощью технологии, называемой *растровым сканированием*. Электронный луч скользит по каждой строке пикселов слева направо и сверху вниз, пока не будут просканированы все строки на экране. Во многих видеодисплеях частота обновления экрана увеличивается за счет *чередования*, то есть сканирования экрана в два прохода — по

четным и по нечетным строкам. Выводимое на экран изображение хранится в *буфере дисплея* (*видеобуфере*) в виде *битовой карты*, содержащей информацию по оси *Z*. Простейшая битовая карта содержит по одному биту на каждый пиксел выводимого на экран изображения. При разрешении экрана 1024×1024 пикселов битовая карта занимает в видеобуфере 1 Мбит памяти. В случае обновления дисплея с частотой 60 раз в секунду данные должны пересылаться со скоростью 60 Мбит/с. Битовая карта современного высококачественного дисплея выделяет по 32 бита для представления каждого пиксела. Для такого дисплея нужен гораздо больший буфер и гораздо более высокая скорость пересылки данных. Обычно для хранения информации о цвете используются только 24 бита из 32, а остальные зарезервированы для обеспечения совместимости с длиной слова процессора и для будущих расширений. Современные системы, в частности операционные системы с оконным интерфейсом, способны воспроизводить одновременно несколько экранных изображений, для хранения которых требуются отдельные видеобуферы.

10.2.2. Плоскопанельные дисплеи

До сих пор доминирующим видом дисплеев были дисплеи на основе ЭЛТ. Однако в последнее время все большую популярность приобретают дисплеи в виде плоских панелей. Они тоньше и гораздо легче ЭЛТ-дисплеев, у них лучше показатели линейности изображения, а в некоторых случаях и более высокое разрешение. Существует несколько типов плоских дисплеев, к числу которых относятся и жидкокристаллические, плазменные и электролюминесцентные панели. Именно благодаря разработке относительно недорогих жидкокристаллических панелей стал возможным выпуск портативных компьютеров.

Жидкокристаллические панели состоят из тонкого слоя жидкокристаллического вещества (жидкости с кристаллическими свойствами), заключенного между двумя прозрачными пластинами. На верхней панели располагаются прозрачные электроды, а задняя панель является зеркальной. Подавая на пластины электрическое напряжение, можно активизировать различные сегменты кристалла, вызвав изменение диффузии или вектора поляризации света. В результате сегменты кристалла будут либо отражать, либо поглощать свет. Изображение формируется в результате того, что луч от источника проходит через сегменты жидкого кристалла и отражается от зеркала. Жидкокристаллические дисплеи используются в часах, калькуляторах, портативных компьютерах и многих других устройствах.

Жидкокристаллические дисплеи бывают двух видов. *Статические* (или *пассивно-матричные*) дисплеи имеют простую структуру, при которой электроды располагаются вдоль оси верхней панели и вдоль перпендикулярной ей оси нижней панели, задавая положение столбцов и строк пикселов. Для освещения конкретного сегмента подается напряжение на два электрода, определяющих положение столбца и строки. В результате создается электрическое поле, под воздействием которого в точке пересечения строки и столбца «включается» жидкий кристалл, и эта точка освещается. Рассматриваемые дисплеи просты в производстве и недороги, но, к сожалению, их качество изображения оставляет желать

лучшего. Освещенная область имеет не очень четкие контуры, так что границы изображений кажутся размытыми. Кроме того, у таких длинных электродов большая емкость, вследствие чего дисплеи имеют большую инерционность. Если быстро перемещать курсор по экрану, за ним будет тянуться заметный след.

Для создания высококачественных дисплеев в каждой точке пересечения строки и столбца размещают по электроду. В результате сокращается время ответа и лучше выделяется освещенная область. Транзисторы располагаются на тонкой пленке одной из пластин. Поэтому дисплеи этого типа именуются *TFT-дисплеями* (Thin-Film Transistor — тонко-пленочный транзистор). Их также называют *активно-матричными*. Активно-матричные дисплеи обычно используются в высококачественных ноутбуках.

Плазменная панель состоит из двух стеклянных пластин, пространство между которыми заполнено газом (как правило, неоном). К каждой пластине подсоединяется ряд параллельных электродов. Электроды двух пластин располагаются под прямым углом друг к другу. Когда на два электрода, находящихся на разных пластинах, подается напряжение, небольшой сегмент газа в точке их пересечения начинает светиться. Плазменные дисплеи обеспечивают высокое разрешение, но они очень дороги. Их используют в тех случаях, когда важно обеспечить высокое качество изображения, но громоздкий ЭЛТ-дисплей при этом не приемлем.

В электролюминесцентных панелях используется тонкий слой фосфора, заключенный между двумя проводящими панелями. Свечение фосфора в нужных точках вызывается электрическим напряжением, приложенным к панелям.

Популярность плоско-панельных дисплеев ограничивается продолжающимся развитием технологий производства ЭЛТ-дисплеев, в которых по-прежнему оптимально сочетаются низкая цена и высокое качество изображения.

10.2.3. Принтеры

Принтеры предназначены для создания печатных копий выходных графических и текстовых данных. В зависимости от механизма печати они подразделяются на ударные и неударные. Работа ударных принтеров построена на основе механического принципа печати, а неударных — на основе оптических, струйных и электростатических технологий.

Неударные принтеры характеризуются очень высокой скоростью печати. К их числу относятся лазерные принтеры, в которых используется та же технология, что и в фотокопировальных аппаратах. Работают они следующим образом. Барабан, поверхность которого покрыта положительно заряженным фотопроводящим материалом, сканируется лазерным лучом. При освещении положительный заряд исчезает. Затем поверх барабана наносится отрицательно заряженный тонер в виде мелкой пудры. Он прилипает к положительно заряженным участкам барабана, создавая изображение страницы, которое затем переносится на бумагу. После печати страницы барабан очищается от остатков тонера.

В струйных принтерах разноцветные чернила попадают на бумагу через маленькие отверстия, создавая цветное изображение. Для выброса чернил используются разные технологии. К примеру, в струйно-пузырьковых принтерах отверстия расположены в небольшой камере, которая нагревается так, что чернила

в ней закипают, образуя маленькие пузырьки, выбрызгивающиеся из отверстия. Когда газ в камере остывает, он создает вакуум, засасывающий в нее из чернильницы новую порцию чернил. Струйные принтеры дешевле лазерных и при этом обладают очень высоким качеством печати.

Большинство принтеров формируют текстовые и графические изображения тем же способом, что и мониторы, то есть в виде множества точек. Это позволяет печатать текст любыми шрифтами и выводить любые графические изображения. Однако качество таких изображений желает лучшего. Поэтому в высококачественных принтерах используется технология, называемая *сглаживанием цветов*. Она заключается в наложении точек в пикселе и цветов в каждой точке, в результате чего получается более качественное изображение и создается видимость большего количества цветов.

Графика и фотографии требуют очень высокого качества печати. С этой целью в струйных принтерах используется технология, называемая термической возгонкой красителя. Подобные принтеры имеют наибольшую стоимость. В них температура нагревания чернил постоянно регулируется, в результате чего изменяется количество чернил, выбрызгиваемых на бумагу. Таким образом изменяется интенсивность цвета точек изображения. Кроме того, используется специальная бумага, на которой чернила смешиваются, в результате чего получаются очень точные цвета.

10.2.4. Графические акселераторы

Для решения многих задач с использованием компьютера необходима высококачественная графика. Изображение такого качества требует вывода на экран большого количества пикселов. Но сначала цвет каждого пиксела нужно вычислить и записать его в видеобуфер. Оттуда информация пересылается в дисплей с такой скоростью, чтобы экран обновлялся по меньшей мере 30 раз в секунду.

Вычисление интенсивности и цвета пикселов может выполняться программным обеспечением. Результирующее изображение слудует записать в видеобуфер, а оттуда переслать на дисплей через шину компьютера. Однако объемы обрабатываемых таким образом данных будут настолько велики, что, если возложить всю их обработку на процессор, у него не останется времени для выполнения других задач. Кроме того, использование шины компьютера для пересылки содержимого видеобуфера на дисплей приведет к тому, что шина также почти полностью будет занята этими данными. Если один пиксел занимает 32 бита, для изображения размером 1024 × 1024 пикселов понадобится 4 Мбайт, и для его пересылки потребуется шина со скоростью передачи не менее 120 Мбайт/с.

В большинстве графических приложений на экран выводятся трехмерные (3D) объекты. В частности, в компьютерных играх создается искусственный трехмерный мир с видеоизображениями, формируемыми программным путем. Для их получения требуются очень сложные вычисления, которые лучше всего выполнять на отдельном специализированном процессоре. Такой процессор, называемый GPU (Graphics-Processing Unit — устройство обработки графики), является основой популярных графических плат, установленных в большинстве персональных

компьютеров. Кроме процессора графическая плата содержит высокоскоростную память объемом от 8 до 64 Мбайт. Эта память используется графическим процессором для выполнения вычислений и хранения результирующего изображения, предназначенного для вывода на экран. Дисплей подключается прямо к графической плате, так что она может обмениваться с ним информацией без помощи шины компьютера. Высококачественные графические платы могут обновлять экран со скоростью от 75 до 200 раз в секунду.

Графический порт

Графическая плата может соединяться с компьютером посредством шины (например, PCI). Однако чаще на материнской плате компьютера имеется соединительный слот, называемый AGP (Accelerated Graphics Port — ускоренный графический порт), специально предназначенный для графической платы. Это 32-раздрядный порт, поддерживающий более высокую скорость пересылки данных, чем шина PCI. Он известен как AGP 1x, 2x, 4x или 8x, где AGP 1x — это исходный стандарт, определяющий передачу данных со скоростью 264 Мбайт/с. Последние версии стандарта AGP поддерживают в несколько раз большие скорости передачи данных, в частности стандартом AGP 8x устанавливается скорость передача данных, равная 2 Гбайт/с.

Графическая обработка

В компьютерной графике трехмерный объект представляется в виде поверхности, состоящей из большого количества маленьких многоугольников (как правило, треугольников). Основной задачей графической обработки является преобразование трехмерного изображения в двухмерное, максимально близкое к тому, каким оно видится человеческим глазом. Для определения *проекции* и *перспективы* объектов требуется вычислять местоположения вершин треугольников, представляющих разные фрагменты изображения. Далее с помощью сложных алгоритмов создания реалистичного изображения вычисляются цвета и тени каждого треугольника. При этих вычислениях учитывается расположение источника света, его отражение от различных поверхностей, тени и т. п. Важной частью данного процесса является формирование определенной текстуры поверхности, например древесных волокон или кирпичной кладки. Текстура обычно задается с помощью элементов, именуемых *текселами* (texel). Отдельные треугольники заполняются текселами, в результате чего создается впечатление текстурной поверхности объекта. Скрытые части изображения удаляются путем *отсечения* (clipping). Последний этап обработки изображения, когда определяется цвет и яркость каждого пиксела, называется *сэмплингом* (sampling), а весь вычислительный процесс, в результате которого трехмерное изображение превращается в набор отправляемых на дисплей пикселов, — *визуализацией* (rendering).

В случае движущихся изображений все эти вычисления повторяются по многу раз в секунду. Чтобы движение на экране было плавным, пикселы изображения должны пересчитываться как минимум 20 раз в секунду, а лучше 30 или 40. Это значение называется *частотой кадров*. Скорость выполнения графической платой описанных вычислений характеризуется ее коэффициентом T&L (Transformations

and Lighting — преобразование и освещение), равным количеству треугольников, для которых видеокарта может выполнить проецирование, отсечение, освещение и самплинг за одну секунду. Как правило, это значение изменяется в пределах от 10 до 30 млн. треугольников в секунду.

В табл. 10.1 приведены характеристики графической платы RADEON VE производства ATI Corp. Похожими возможностями обладает графический процессор GeForce 2 MX производства *n*Vidia Corp. Это примеры популярных плат для персональных компьютеров. В профессиональных системах используются более мощные платы с расширенными возможностями. А в ближайшем будущем в этой быстро развивающейся области компьютерной индустрии ожидается появление еще более мощных процессоров.

Таблица 10.1. Графическая плата RADEON VE

Компонент	Описание
Микросхема GPU	RADEON VE
Шина	AGP 4x
Память	До 64 Мбайт, DDR SDRAM
Цвет	32 бита, включая 8 бит, зарезервированных для будущего использования
Число пикселов	2048 × 1536
Коэффициент T&L	30 млн треугольников в секунду
Частота обновления экрана	От 75 до 200 раз в секунду в зависимости от установленного разрешения
Дополнительные возможности	Поддержка TV, VCR, DVD, HDTV и MPEG 2

Программное обеспечение графических плат

Графические платы предназначены для реализации множества сложных функций. Чтобы их использовать, нужно иметь специальное программное обеспечение, разработанное для конкретной платы. В этой области очень мало стандартов, и рынок открыт для конкуренции. Таким образом, для улучшения качества изображения недостаточно просто установить в компьютер лучшую графическую плату. Требуется специальное программное обеспечение. Очевидно, что назрела необходимость в разработке стандартов программных интерфейсов приложений (Application Programming Interface, API), позволяющих создавать аппаратно-независимое программное обеспечение. И такие стандарты уже начинают появляться. Когда они получат достаточное распространение, программное обеспечение, интенсивно использующее возможности графики (например, компьютерные игры), сможет корректно работать с графическими платами разных производителей. Примером такого стандарта является OpenGL (Open Graphics Language — открытая графическая библиотека). Ему и подобным стандартам, связанным с различными аспектами обработки графики, соответствует все больше графических плат.

10.3. Последовательные коммуникационные соединения

Такие устройства, как мышь и клавиатура, подключаются прямо к компьютеру, и для этого, как правило, используется последовательное соединение. Другие устройства, в частности принтеры и сканеры, могут подключаться к компьютеру либо непосредственно, либо через коммуникационную сеть, чтобы их могли совместно использовать несколько пользователей. Поскольку во многих компьютерных приложениях важную роль играет Интернет, компьютеры часто соединяются с ним либо непосредственно, либо через коммутируемые телефонные линии.

Далее в этой главе описываются типичные схемы последовательных коммуникационных соединений. Мы начнем с рассмотрения некоторых основных идей и понятий в этой области.

Модуляция и демодуляция

В электронных схемах бит кодируется электрическим сигналом, который может принимать одно из двух значений. Каналы передачи данных, по которым пересылаются такие сигналы, называются *узкополосными*. Альтернативой подобного кодирования нулей и единиц может выступать модуляция синусоидального *несущего сигнала*. Для передачи таких сигналов используются *широкополосные* каналы. Предположим, что частота сигнала может принимать одно из двух значений: f_1, представляющее логическое значение 0, и f_2, представляющее логическое значение 1. В этом случае говорят, что используется частотная модуляция, ЧМ (Frequency Modulation, FM). Существует и множество других схем модуляции. При фазовой модуляции, ФМ (Phase Modulation, FM) изменяется фаза несущего сигнала, а при амплитудной модуляции, АМ (Amplitude Modulation, AM) — его амплитуда. При квадратурной амплитудной модуляции, КАМ (Quadrature Amplitude Modulation, QAM) изменяются и амплитуда, и фаза несущего сигнала, благодаря чему возможны не две, а четыре комбинации значений. Таким образом, передаваемый сигнал может представлять два бита информации.

При передаче данных параметры несущего сигнала изменяются скачкообразно. Такой способ модуляции сигнала называется манипуляцией (keing). Соответственно различают частотную, ЧМн (Frequency-Shift Keying, FSK); фазовую, ФМн (Phase-Shift Keying, PSK); амплитудную, АМн (Amplitude-Shift Keying, ASK) и квадратурную амплитудную, КАМн (Quadrature Amplitude-Shift Keying, QASK) манипуляции.

Конфигурация сигнала, передаваемая в течение одного тактового периода, именуется *символом*. При частотной модуляции используются два символа, представленных синусоидальными сигналами с частотами f_1 и f_2. При квадратурной модуляции применяются четыре символа, определяемых амплитудой и фазой сигнала. Количество символов, передаваемых за одну секунду, составляет *бод* (baud). Его же можно определить как количество изменений состояния сигнала в секунду. Количество бодов в секунду равняется количеству битов в секунду только в том случае, если используется двоичная схема модуляции, как в описанном выше примере частотной модуляции. Для квадратурной модуляции значение скорости

передачи данных в битах вдвое больше значения в бодах, поскольку каждый символ представляет два бита информации. Существуют схемы модуляции, где используется 8, 16 или более символов. В системе с 16 символами для кодирования одного символа необходимо 4 бита, а скорость передачи данных в битах вчетверо больше скорости передачи в бодах.

Как показано на рис. 10.2, на концах коммуникационного канала связи располагается устройство, называемое *модемом* (МОдулятор-ДЕМодулятор), которое выполняет преобразование сигналов. На этом рисунке изображен компьютер, соединенный с сетевым сервером. Соединение может быть постоянным или временным, устанавливаемым через телефонную линию (коммутируемым).

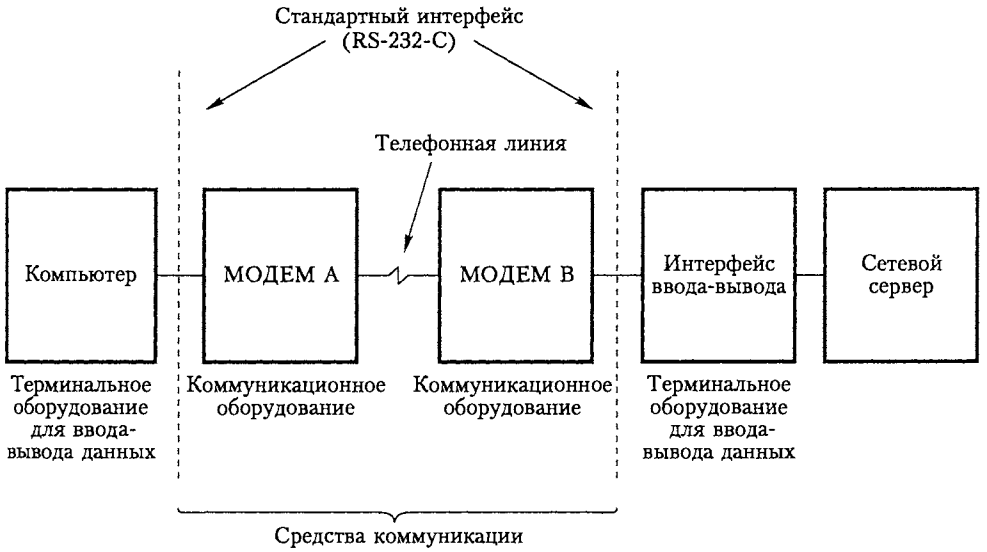


Рис. 10.2. Удаленное подключение к сети

Синхронизация

Последовательное соединение означает, что данные пересылаются по одному биту за единицу времени. Для этого необходимо, чтобы передающее и принимающее устройства использовали для интерпретации отдельных битов одну и ту же тактовую информацию. Когда взаимодействующие устройства расположены в непосредственной близости друг от друга, скажем в одном помещении, и имеется несколько сигнальных линий, тактовый сигнал может передаваться одновременно с данными. Однако для устройств, находящихся на большом расстоянии друг от друга, такая технология пересылки неприемлема, поскольку имеется всего один сигнальный канал. Но даже в случае наличия второго канала задержки при пересылке данных и тактового сигнала могут различаться. Поэтому тактовая информация и данные все равно кодируются вместе и передаются по одному каналу. Существует множество разнообразных схем кодирования, позволяющих принимающему устройству декодировать полученный сигнал и правильно выделять данные и тактовую информацию.

Реализация последовательной передачи данных осуществляется одним из двух способов. Если скорость пересылки не превышает нескольких десятков килобайтов в секунду, может применяться простая схема, в которой приемник и передатчик используют независимые тактовые сигналы одинаковой номинальной частоты. При этом не гарантируется, что два тактовых сигнала будут в точности совпадать по частоте и фазе. Поэтому такая схема называется *асинхронной*.

Для передачи данных на более высокой скорости применяется *синхронная* схема: приемник определяет используемую передатчиком тактовую частоту по расположению переходов получаемого сигнала и соответственно настраивает свою тактовую частоту. В результате тактовые сигналы приемника и передатчика синхронизированы и передаваемые данные всегда распознаются правильно. Для кодирования тактовой информации, передаваемой по синхронным каналам связи, могут быть задействованы разные технологии. Они различаются способом использования полосы пропускания канала и максимальной скоростью пересылки данных.

Дуплексное и полудуплексное соединения

Коммуникационное соединение может функционировать по одной из трех следующих схем:

- *симплексное* соединение — поддерживает передачу данных только в одном направлении;
- *полудуплексное* соединение — позволяет передавать данные в обоих направлениях, но не одновременно;
- *дуплексное* соединение — позволяет одновременно передавать данные в двух направлениях.

Симплексная конфигурация полезна лишь в тех случаях, когда на одном из концов канала связи располагается только принимающее или только передающее устройство, но не оба сразу. Эта конфигурация применяется редко. Обычно для передачи данных используется дуплексная или полудуплексная конфигурация, а выбор между ними зависит от требуемого соотношения экономичности и скорости передачи данных.

Самое простое электрическое соединение для передачи данных состоит из двух проводов, по которым данные могут пересылаться только в одном направлении. Речь идет о симплексном соединении. Полудуплексное соединение создается путем расположения на разных концах соединения двух переключателей, соединяемых поочередно с передающим и принимающим устройствами. Когда завершается пересылка данных в одном направлении, выполняется переключение на другое направление. За управление переключателями отвечают устройства, расположенные на концах соединения.

Дуплексное соединение может состоять из четырех проводов, по два на каждое направление передачи. Возможно также использование двухпроводного соединения с двумя непересекающимися полосами частот. Две полосы частот создают два канала передачи, по одному на каждое направление. В качестве альтернативы можно использовать общую полосу частот, если на концах соединения будет работать устройство, называемое *гибридом* (hybrid). Гибрид разделяет сигналы, передаваемые в разных направлениях, чтобы они не влияли друг на друга. Такая схема передачи данных применяется в коммутируемых телефонных соединениях.

В случае синхронной полудуплексной связи при изменении направления передачи данных происходит задержка, поскольку передающий модем должен передать инициализационную последовательность сигналов, чтобы принимающее устройство адаптировалось к новому состоянию канала. Длительность этой задержки зависит от параметров модема и канала связи и лежит в диапазоне от нескольких до сотен миллисекунд.

Все вышесказанное относится к характеристикам линий связи и модемов. Другими важными факторами, от которых зависит выбор между полудуплексным и дуплексным соединением, является природа трафика данных и способ реакции системы на ошибки при передаче. Мы обсудим только первый из этих факторов.

При работе со многими приложениями необходимо, чтобы компьютер получил входные данные, обработал их и вернул выходные данные. Требованиям таких приложений прекрасно удовлетворяет полудуплексная связь. Однако если выполняется частый обмен короткими сообщениями, задержка на переключение направления передачи данных начинает существенно отражаться на скорости работы. Поэтому во многих приложениях используют дуплексные соединения, хотя данные никогда не передаются в обоих направлениях одновременно.

В некоторых ситуациях очень удобна одновременная передача данных в двух направлениях. Например, в системе на рис. 10.2 пользователь может непосредственно взаимодействовать с сетевым сервером, используя свой компьютер в качестве терминала. Каждый вводимый с клавиатуры символ должен также отображаться на дисплее компьютера. Это может осуществляться локально самим компьютером или удаленно сетевым сервером. Если это делается удаленно, необходимы средства автоматического контроля принимаемых данных, гарантирующие отсутствие ошибок. При использовании в такой ситуации полудуплексной связи пересылку очередного введенного символа придется откладывать до получения предыдущего символа. В случае применения дуплексной связи этого ограничения не будет. Еще одним примером ситуации, когда удобна дуплексная связь, является соединение узлов высокоскоростной компьютерной сети. Сообщения, пересылаемые через конкретное соединение в двух направлениях, могут быть никак не связаны между собой. Поэтому их можно передавать одновременно.

10.3.1. Асинхронная передача

Простейшей схемой последовательной связи является асинхронная передача с использованием стартстопной технологии. Для обеспечения надежной синхронизации данные объединяются в маленькие группы по 6 или 8 бит с четко выделенным началом и концом. Типичным примером является пересылка буквенно-цифровых символов в виде 8-разрядных кодов (рис. 10.3). Когда данные не передаются, линия, соединяющая приемник с передатчиком, находится в состоянии 1. Пересылке символа предшествует пересылка бита с логическим значением 0, такой бит называется стартовым. За ним следуют 8 бит данных и один или два стоповых бита. Стоповые биты имеют логическое значение 1. Стартовый бит сообщает приемнику о начале передачи данных. Его передний фронт используется для синхронизации тактового сигнала приемника с тактовым сигналом передатчика.

При непрерывной передаче данных стоповые биты в конце символа отделяют его от следующих символов. Когда передача данных прекращается, после передачи последнего стопового бита линия остается в состоянии 1. За вставку и удаление стоповых и стартовых битов отвечают схемы приемника и передатчика.

Рис. 10.3. Асинхронная последовательная пересылка символов

Чтобы обеспечить правильную синхронизацию со стороны получателя, его тактовый сигнал, управляющий передачей данных, нужно сгенерировать на основе локального тактового сигнала, имеющего гораздо более высокую частоту (обычно она больше в 16 раз). Это означает, что на каждый бит передаваемых данных приходится 16 тактовых импульсов. Такая тактовая частота используется для приращения счетчика по модулю 16, который сбрасывается в 0 при появлении переднего фронта стартового бита. Когда счетчик принимает значение 8, это означает, что достигнута середина стартового бита. Приемник проверяет значение стартового бита, чтобы убедиться, что это действительно правильный стартовый бит, а затем счетчик снова сбрасывается в 0. С этого момента (который приходится примерно на середину передачи бита) значение входящего сигнала проверяется всякий раз, когда счетчик достигает значения 16. И пока относительное расположение битов передаваемого символа не сместится более чем на половину такта, приемник интерпретирует биты кода символа корректно.

Серийно выпускаемое оборудование поддерживает разные скорости передачи данных — от 300 до 56000 бит в секунду. Стартстопная передача используется для коротких соединений, таких как соединение между компьютером и модемом (рис. 10.2). Для больших расстояний эта схема может использоваться только на малых скоростях передачи данных. Высокоскоростные модемы работают на основе схем синхронной передачи, описанных в следующем разделе.

При передаче символьных данных каждый символ представляется 7-битовым ASCII-кодом (см. приложение Д), занимающим биты от 0 до 6 (рис. 10.3). Самый старший бит передаваемого байта обычно устанавливается в 0. В качестве альтернативы он может использоваться для контроля четности, помогающего выявлять ошибки передачи. Бит четности представляет собой сумму группы битов по модулю 2. Он равен 1, если передаваемые данные содержат нечетное количество

единиц, и 0 в противном случае (его значение устанавливается передатчиком). Если в результате ошибки в ходе пересылки изменится один бит данных, получатель обнаружит, что бит четности не соответствует битам данных.

Набор символов ASCII включает буквы, цифры и специальные символы, такие как $, + и >. Кроме того, в нем имеется несколько непечатных символов, например EOT (End Of Transmission — конец передачи) и CR (Carriage Return — возврат каретки). Эти символы могут использоваться для запроса определенных действий, в особенности при передаче сообщений удаленному компьютеру и получении сообщений от него.

10.3.2. Синхронная передача

В описанной выше стартстопной схеме с целью синхронизации тактирования приема данных используется позиция перехода от 1 к 0 в начале стартового бита (рис. 10.3). Поэтому такая схема может применяться только в тех случаях, когда скорость передачи данных невысока и условия их передачи по линии позволяют сохранять прямоугольную форму сигнала. На более высоких скоростях и больших расстояниях передачи данных происходит значительное искажение сигнала. На рис. 10.4, где показано несколько наложенных друг на друга импульсов битов, демонстрируется, как может изменяться форма сигнала. Искажение сигнала обусловлено помехами на линии и параметрами передающего оборудования, наводками от внешних устройств, дрожанием (случайными изменениями положения передаваемого сигнала) и т. д. Представленная на рис. 10.4 диаграмма называется *глазковой диаграммой* передачи (из-за формы сигнала). Для определения центральной точки глазковой диаграммы, в которой единица и ноль располагаются на максимальном расстоянии друг от друга, в приемнике используются сложные схемы кодирования и декодирования сигналов. Измерение полученного сигнала лучше всего производить именно в этой точке.

Рис. 10.4. Диаграмма передачи сигнала

При синхронной передаче данные пересылаются блоками, состоящими из нескольких сотен или тысяч битов. Начало каждого блока отмечается соответствующим кодом, а данные в блоке организуются в соотвествии с определенными правилами. Для выполнения таких операций, как передача и получение данных о несущей частоте и синхронизация, модему требуется достаточно много времени. Некоторые модемы перед началом передачи данных выполняют еще и процедуру адаптации к свойствам линии связи.

Сетевые соединения — ADSL

За последние годы, и в особенности в связи с широким распространением World Wide Web, все больше возрастает потребность подключения домашних и офисных компьютеров к Интернету через высокоскоростные линии связи. До недавнего времени производительность модемов была для этого явно недостаточной.

Стандартный модем преобразовывает цифровые сигналы в аналоговые с использованием частот голосового диапазона (до 4 кГц), передаваемых по телефонной линии. Когда один компьютер при помощи такого модема взаимодействует с другим, линия недоступна для обычных телефонных звонков. Но что еще важнее, скорость передачи данных ограничена несколькими десятками килобитов в секунду. Это гораздо меньше той скорости, которая необходима для подключения к удаленному серверу или к Интернету.

Традиционные телефонные технологии используют лишь малую часть реальной пропускной способности телефонной линии. В зависимости от расстояния и свойств линии современные методы коммуникации позволяют передавать по применяемой в телефонной системе витой паре до 50 Мбит данных в секунду. Разработано множество схем использования этой дополнительной полосы пропускания путем непосредственной передачи оцифрованной информации между пользователем и центральным офисом телефонной компании. Соединение между пользователем и центральным офисом получило название *абонентской линии*. Поэтому соединение для цифровой передачи данных по телефонным линиям называется *цифровой абонентской линией* (Digital Subscriber Line, DSL).

Уровень взаимодействия компьютеров в значительной степени зависит от соответствия используемого оборудования тем сервисам, которые предоставляются разными компьютерными компаниями, производителями модемов и провайдерами сетевых услуг. Очень важно, чтобы все стороны, от которых так или иначе зависит процесс взаимодействия, придерживались некоторых общих стандартов. В области DSL уже есть несколько таких стандартов. Это SDSL (Symmetric DSL — симметричная цифровая абонентская линия), HDSL (High speed DSL — высокоскоростная цифровая абонентская линия) и ADSL (Asymmetric DSL — асимметричная цифровая абонентская линия). При подключении домашних компьютеров к Интернету наиболее часто применяется стандарт ADSL. Поэтому мы вкратце расскажем о его основных особенностях.

Слово «асимметричная» в названии стандарта ADSL указывает на разницу в скорости передачи данных в прямом и обратном направлениях. Большая часть информации, посылаемой от компьютера к серверу (прямое направление), состоит из вводимых пользователем данных. Для ее передачи достаточно низкоскоростного соединения. С другой стороны, информация, передаваемая от сервера к пользователю (обратное направление), такая как выводимые на экран пользователя изображения, должна пересылаться на высокой скорости, чтобы в работе пользователя не было больших задержек. Поэтому скорость передачи в обратном направлении в ADSL значительно выше скорости передачи в прямом направлении.

Для создания нескольких коммуникационных каналов в ADSL используются различные частотные полосы и специальная технология, называемая мультиплексной передачей с временным разделением. Один из этих каналов выделен для обычного телефонного обслуживания, а остальные предназначены для пересылки данных в прямом и обратном направлениях. На рис. 10.5 показана типичная схема соединения ADSL. Информация между абонентом и центральным офисом телефонной компании, предоставляющим услуги соединения с Интернетом по технологии ADSL, передается по обычной витой паре. На ее концах находятся разделяющие устройства, называемые сплиттерами, которые отделяют трафик данных от

голосового сигнала. Со стороны абонента данные передаются в компьютер через любое подходящее соединение, например Ethernet или USB. Голосовые сигналы направляются в телефон. В центральном офисе данные передаются в Интернет через маршрутизатор, а голосовые сигналы отсылаются на телефонный коммутатор. (Маршрутизатор — это коммутирующее устройство, используемое для управления трафиком в сетях данных.) При такой схеме компьютер может быть все время подключен к Интернету без необходимости набора номера, причем обычные коммутируемые телефонные услуги остаются доступными.

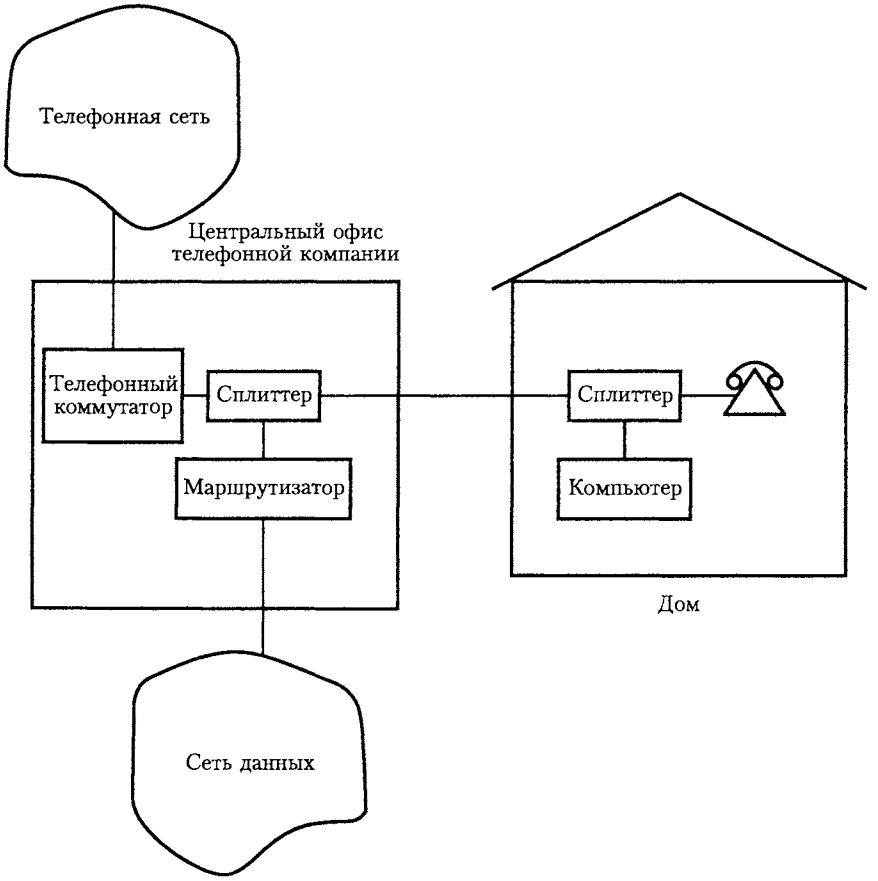


Рис. 10.5. Соединение ADSL

Кабельные модемы

Альтернативным средством соединения домашнего компьютера с Интернетом является кабельный модем. Вместо телефонных линий такой модем использует для связи телевизионные кабели. Коаксиальный кабель, применяемый в кабельном телевидении, имеет более широкую полосу пропускания, чем обычная витая пара. Поэтому максимальная скорость передачи данных при помощи кабельного

модема гораздо выше скорости DSL-соединений. Правда, в кабельном телевидении все абоненты в пределах некоторой области подключаются к одному кабелю, поэтому полоса пропускания кабеля разделяется между всеми работающими абонентами. А вся полоса пропускания кабеля может быть предоставлена в распоряжение одного абонента лишь в том случае, если никто из остальных абонентов, подключенных к этому же кабелю, в данный момент «не активен». Типичная схема кабельного соединения показана на рис. 10.6.

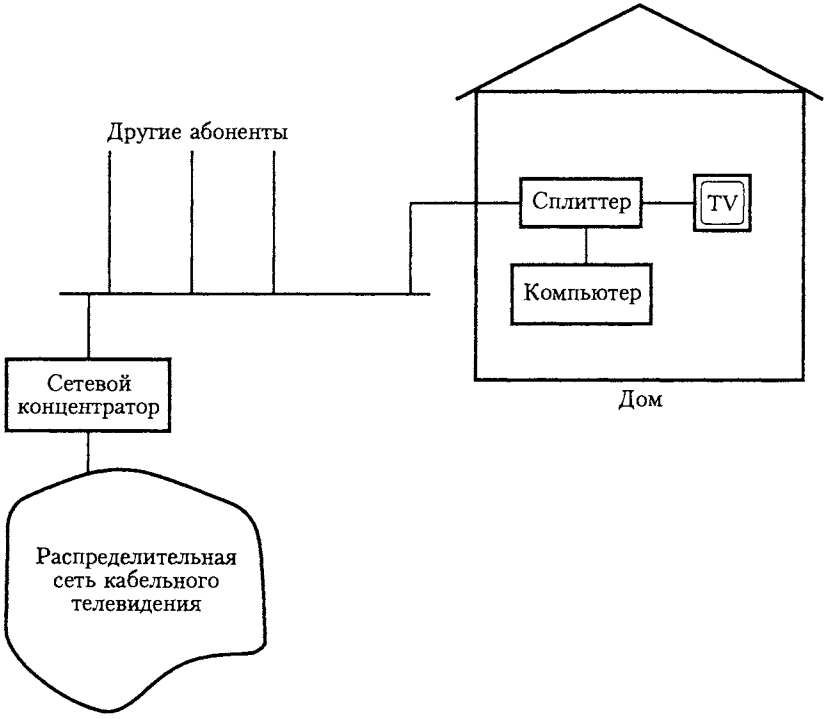


Рис. 10.6. Соединение при помощи кабельного модема

Максимальная скорость, которую кабельный модем может обеспечить при работе одного пользователя, определяется провайдером сетевых услуг. Как правило, она находится в диапазоне от 600 Кбит/с до 10 Мбит/с.

10.3.3. Стандартные коммуникационные интерфейсы

Стандартный коммуникационный интерфейс определяет способ соединения двух устройств. Одним из широко распространенных коммуникационных стандартов является стандарт RS-232-C, разработанный ассоциацией EIA (Electronics Industries Association — Ассоциация электронной промышленности). За пределами Северной Америки этот стандарт известен как рекомендация CCITT (Comite Consultatif Internationale de Telegraphique et Telephonique — Международный консультативный комитет по телеграфии и телефонии, МККТТ) под номером V24.

Этот стандарт устанавливает интерфейс между коммуникационными устройствами, такими как модемы, и терминальным оборудованием, в частности компьютерами. Интерфейс RS-232-C состоит из 25 элементов, описанных в табл. 10.2.

Таблица 10.2. Сигналы стандарта EIA RS-232-C (рекомендация V24 CCITT)

EIA	CCITT	Номер контакта[1]	Описание
AA	101	1	Защитное заземление (Protective ground)
AB	102	7	Сигнальное заземление (Signal ground)
BA	103	2	Передаваемые данные (Transmitted Data)
BB	104	3	Принимаемые данные (Received Data)
CA	105	4	Запрос для передачи (Request to send)
CB	106	5	Сброс для передачи (Clear to Send)
CC	107	6	Готовность данных (Data Set Ready)
CD	108.2	20	Готовность выходных данных (Data Terminal Ready)
CE	125	22	Индикатор вызова (Ring Indicator)
CF	109	8	Детектор принимаемого с линии сигнала (Received line signal detector)
CG	110	21	Детектор качества сигнала (Signal quality detector)
CH	111	23[4]	Выбор скорости передачи данных (Data signal rate selector) от DTE[2] к DCE[3]
CI	112	23[4]	Выбор скорости передачи данных (Data signal rate selector) от DCE[3] к DTE[2]
DA	113	24	Тактирование сигнального элемента передатчика (DTE[2])
DB	114	15	Тактирование сигнального элемента передатчика (DCE[3])
DD	115	17	Тактирование сигнального элемента приемника (DCE[3])
SBA	118	14	Вторичные передаваемые данные
SBB	119	16	Вторичные принимаемые данные
SCA	120	19	Вторичный запрос для передачи
SCB	121	13	Вторичный сброс для передачи
SCF	122	12	Вторичный детектор принимаемого с линии сигнала

[1] Контакты 9 и 10 используются для тестирования, а контакты 11, 18 и 25 — запасные.
[2] Терминальное оборудование.
[3] Коммуникационное оборудование.
[4] Имя сигнала на этом контакте зависит от направления сигнала.

Давайте рассмотрим простой, но типичный пример. Предположим, что схема, показанная на рис. 10.2, реализуется при помощи коммутируемого телефонного соединения. Используемые для соединения модемы могут формировать сигналы о начале и завершении сеанса связи, передавать тоновые сигналы набора номера и обнаруживать входящий сигнал вызова. Они работают на основе описанной выше

схемы дуплексной передачи с частотной модуляцией и поддерживают два канала передачи — по одному для каждого направления. Один канал использует частоты 1275 и 1075 Гц, а второй — 2225 и 2025 Гц, представляющие логические значения 0 и 1 соответственно.

На рис. 10.7 приведена последовательность логических сигналов, необходимых для установки подключения. Этот процесс включает следующие этапы.

1. Когда сетевой сервер готов к приему звонка, он устанавливает сигнал готовности выходных данных (CD) в 1.
2. Модем B выполняет мониторинг телефонной линии и, обнаружив сигнал входящего звонка, сообщает об этом серверу путем установки сигнала индикатора вызова CE в 1. Если в этот момент CD = 1, модем автоматически предает сигнал о снятии трубки. Затем он устанавливает сигнал готовности модема CC в 1.
3. Сервер указывает модему B на необходимость начать передачу значения частоты, используемой для представления единицы (2225 Гц), для чего он устанавливает сигнал запроса для передачи CA в 1. Передав указанную частоту, модем B устанавливает сигнал сброса для передачи CB в 1. Определив эту частоту, модем A устанавливает сигнал детектора принимаемого с линии сигнала CF в 1.
4. Компьютер устанавливает CA в 1. Модем A передает сигнал с частотой 1275 Гц и устанавливает CB и CC в 1. Когда модем B определяет, что частота равна 1275 Гц, он устанавливает CF в 1.
5. Теперь между сервером и компьютером установлена дуплексная связь, которая может использоваться для передачи данных в любом направлении. Для этих целей применяются контакты интерфейса BA (передаваемые данные) и BB (получаемые данные); все остальные сигналы интерфейса остаются неизменными.
6. Когда пользователь заканчивает связь, сервер устанавливает сигналы запроса о готовности выходных данных и возможности их передачи, CA и CD, в 0, и в ответ модем B сбрасывает сигнал 2225 Гц и отсоединяется от линии. Сигналы CB, CF и CC устанавливаются модемом B в 0. Когда модем A обнаруживает отсутствие сигнала на линии, он устанавливает сигнал детектора принимаемого с линии сигнала CF в 0.
7. Модем A сбрасывает сигнал 1275 Гц, устанавливает CB и CC в 0 и передает сигнал о завершении связи.
8. Сервер устанавливает сигнал готовности выходных данных CD в 1 и ждет поступления нового звонка.

Применяемая модемами процедура установки соединения предполагает обмен сообщениями, с помощью которых две стороны договариваются о таких параметрах, как схема кодирования, скорость передачи, размер блоков данных и т. д. Интерфейс RS-232-C позволяет установить последовательное соединение между любыми двумя цифровыми устройствами. Интерпретация отдельных сигналов, в частности таких как CA и CD, зависит от функциональных возможностей

устройств. Когда эти сигналы не нужны, они просто игнорируются обоими устройствами. В большинстве случаев используется не более 9 сигналов из приведенных в табл. 10.2.

Шаг	Компьютер	Сигналы интерфейса	Модем А	Модем В	Сигналы интерфейса	Сервер
1				Включение автоматического ответа	CD	← 1
2	Набираемые цифры →			1 → Снимает трубку 1 →	CE CC	
3		CF	← 1	← 2225 Гц 1 →	CA CB	← 1
4	1 →	CA CB CC	1275 Гц → ← 1 ← 1	1 →	CF	
5	Выходные данные ← Входные данные →	BB BA	← Данные 1275/1075 Гц →	← 2225/2025 Гц Данные →	BA BB	← Выходные данные → Входные данные
6		CF	← 0	Снимает 2225 Гц и отключается 0 → 0 → 0 →	CA CD CF CC CB	← 0 ← 0
7	(0 →) Конец связи	CA CB CC	1275 Гц ← 0 ← 0			
8					CD	← 1

Рис. 10.7. Стандартная последовательность сигналов интерфейса RS-232-C

10.4. Резюме

В этой главе рассматривались принципы работы основных устройств ввода и вывода. Устройства ввода-вывода являются одной из важнейших частей любой компьютерной системы, поскольку именно через них информация вводится в компьютер, а результаты его работы возвращаются пользователю. За последние годы было разработано множество новых, более удобных и эффективных устройств, а технологии производства традиционных устройств были значительно усовершенствованы. Благодаря этому высококачественные устройства вывода для персональных компьютеров теперь можно приобрести за сравнительно невысокую цену. Все больше расширяется диапазон устройств ввода, пополнившийся цифровыми фотоаппаратами и видеокамерами, а также рядом других портативных устройств.

В настоящей главе речь шла о коммуникационных технологиях, с помощью которых осуществляется связь между компьютерами. В частности, были рассмотрены основы последовательной связи. Такая связь часто используется для соединения компьютеров между собой и с устройствами ввода-вывода. Мы вкратце рассмотрели организацию высокоскоростных соединений с Интернетом посредством телефонных линий и сетей кабельного телевидения. Доступность средств связи, компьютеров и других цифровых устройств, а также разработка и распространение соответствующих технологий невероятно расширили возможности применения компьютерной техники не только на производстве или, скажем, в офисах, но и дома. Объединение этих двух технологических областей — компьютерной техники и средств коммуникации — положило начало новой эре информационных технологий.

Упражнения

10.1. Чтобы изображение на экране видеодисплея не дрожало, его следует обновлять по меньшей мере 30 раз в секунду. Время, необходимое для подсветки каждой точки в процессе сканирования экрана, равняется 1 мкс. По истечении этого периода электронный луч гасится и перемещается к следующей точке. На перемещение луча от точки к точке в среднем уходит 3 мкс. По причинам, связанным с потреблением энергии, луч не может быть включен дольше, чем на 1/10 (10 %) от этого времени. Определите максимальное количество точек экрана, которые он может подсветить.

10.2. Рассмотрим коммуникационный канал, для которого используется восемь значений сигнала, а не два, как для обычного двоичного канала. Если канал пропускает данные со скоростью 9600 бод, какова его пропускная способность в битах в секунду?

10.3. Имеются следующие компоненты:

- 6-битовый двоичный счетчик с двумя входами (для тактового сигнала и очистки) и шестью выходами;
- 3-битовый сдвиговый регистр с последовательным входом и параллельным выходом;
- генератор тактового сигнала с частотой, в восемь раз превышающей частоту входных данных;
- логические вентили и D-триггеры со входами установки и очистки.

Разработайте схему для загрузки 3 бит последовательных данных со входной линии данных в сдвиговый регистр. Предполагается, что данные имеют такой формат, как показано на рис. 10.3, но число битов данных равно не 8, а 3. Необходимо, чтобы в разработанной вами схеме было два выхода, *A* и *B*, первоначально устанавливаемых в 0. Если за битами данных обнаруживается стоповый бит, выход *A* должен быть установлен в 1. В противном случае выход *B* должен быть установлен в 1. Объясните, как работает созданная вами схема.

10.4. Для асинхронной связи между двумя компьютерами используется старт-стопная схема с одним стартовым и одним стоповым битами и задается скорость передачи данных 38,8 Кбит/с. Какова реальная скорость передачи данных для каждого из компьютеров?

10.5. Каждый передаваемый символ проверяется на нечетность. Обратившись к приложению Д, приведите 8-разрядные коды, передаваемые для символов A, P, = и 5.

10.6. Имеется модем, соединенный с компьютером посредством интерфейса RS-232-C. Управляющие сигналы этого интерфейса доступны компьютеру через 16-разрядный регистр, показанный на рис. У10.1. Бит изменения состояния b_{15} устанавливается в 1 в тех случаях, когда изменяется состояние бита b_{12} или b_{13} либо когда b_{14} установлен в 1. При обращении процессора к этому регистру бит b_{15} очищается. Напишите программу для одного из представленных в главе 3 процессоров, которая реализовывала бы управляющую последовательность сигналов, необходимую для установки телефонного соединения в соответствии с шагами 1–4 на рис. 10.7.

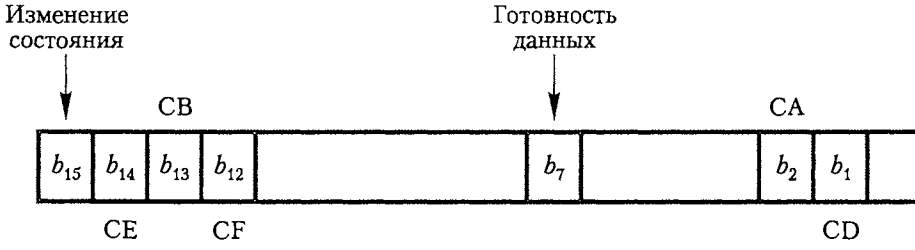


Рис. У10.1. Структура регистра ввода-вывода для интерфейса модема из упражнения 10.6

Глава 11

Семейства процессоров

- Семейства процессоров ARM, Sun SPARC, Compaq Alpha и Intel IA-64, обладающие системами команд с архитектурой RISC
- Семейства процессоров Motorola 680X0/ColdFire и Intel IA-32, обладающие системами команд с архитектурой CISC
- Семейство процессоров Hewlett-Packard HP3000, поддерживающих команды для обработки стековых структур данных

В главе 2 рассказывалось, как реализуются базовые концепции программирования на уровне языка ассемблера и для чего нужны различные машинные команды и режимы адресации. В главе 3 указанные концепции рассматривались на примере архитектуры систем команд трех семейств процессоров — ARM, Motorola 68000 и Intel IA-32. На данном этапе мы продолжим обсуждение этих систем, но теперь сосредоточим внимание на функциях разных членов семейств процессоров. Процессоры семейства ARM имеют архитектуру RISC, а процессоры семейств Motorola и Intel — архитектуру CISC. Вместе с тем в настоящей главе описываются: архитектура PowerPC, относящаяся к типу RISC и конкурирующая с Intel IA-32, архитектуры Sun SPARC, Compaq Alpha и Intel IA-64. Рассматриваемые ниже 64-разрядные процессоры предназначены в первую очередь для использования в высокопроизводительных рабочих станциях и серверах. Наконец, мы познакомимся с уникальным подходом к организации системы команд процессора, согласно которому для хранения операндов применяется стековая структура данных. Такой подход используется в процессорах Hewlett-Packard HP3000 и интересен скорее с точки зрения истории, а не практики.

В главах 2, 3 и 7 дан обзор различных подходов к организации процессоров, в основе которых лежит как архитектура RISC, так и CISC. В данной главе обсуждаются важнейшие особенности каждого из этих подходов. Теоретический материал подкрепляется примерами, демонстрирующими, в каких процессорных семействах они используются. Системы команд CISC-процессоров включают множество мощных команд, которые непосредственно реализуют ряд операций, свойственных языкам высокого уровня, и поддерживают структуру управления потоком выполнения программы. Процесс выполнения таких команд может быть довольно сложным. Концепция CISC позволяет значительно сократить количество машинных команд, необходимых для реализации команд языка высокого уровня. Однако это возможно лишь в том случае, если сложные машинные команды могут быть выполнены быстро и эффективно, что на практике часто составляет проблему и требует увеличения площади микросхемы процессора. Кроме того, система команд типа CISC усложняет задачу оптимизирующих компиляторов.

Концепция RISC основана на использовании относительно простых команд и на первый взгляд может показаться менее эффективной, чем CISC, поскольку решение конкретной вычислительной задачи требует расширенного набора RISC-команд. Однако RISC-команды в большей степени подходят для конвейерного выполнения, в связи с чем скорость их обработки выше. Главным преимуществом RISC-архитектуры является то, что она более эффективно используется оптимизирующими компиляторами. Еще одно ее достоинство связано с технологиями производства СБИС (сверхбольших интегральных схем). Поскольку для обработки команд в RISC-процессоре требуется меньшая площадь микросхемы, остается больше места для регистров и кэша. В результате сокращается количество обращений к данным и командам, хранящимся вне микросхемы процессора, и скорость выполнения программ значительно повышается.

Принимая во внимание все эти факторы, можно заключить, что оба метода пригодны для создания конкурентоспособных коммерческих продуктов. Несмотря на это в современных компьютерах, разрабатываемых с начала 1990-х годов, используются главным образом системы команд типа RISC.

Считаем нужным еще раз подчеркнуть, что на скорость выполнения команд процессором влияет множество факторов, а архитектура системы команд — это только один из них. В главе 8 рассказывалось об использовании дублирующихся конвейерных функциональных блоков процессора. Кроме того, важную роль играют оптимизирующие компиляторы, транслирующие программы на языке высокого уровня в эффективный машинный код.

11.1. Семейство процессоров ARM

В первой части главы 3 рассматривалась архитектура ARM в качестве примера системы команд типа RISC. Процессоры ARM предназначены главным образом для встроенных систем. Поэтому они должны иметь невысокую цену и потреблять мало энергии. Многие устройства, например мобильные телефоны, питаются от батарей с напряжением от 1 до 3 В. По сравнению с высокопроизводительными процессорами Intel Pentium, рассчитанными на рынок персональных компьютеров, процессоры ARM имеют более простую структуру и содержат гораздо меньше транзисторов. Далее мы поговорим о различных реализациях системы команд ARM и обсудим ряд важных вопросов.

С момента появления архитектуры ARM (середина 1980-х годов) и до 2000 года было разработано пять версий системы команд ARM — от v1 до v5. В главе 3 вы познакомились с версией v3 и реализующим ее процессором ARM7, созданным в середине 1990-х годов. Материал о различных моделях этого процессора и их важнейших характеристиках предлагается вашему вниманию в следующих разделах.

Версии v1 и v2 поддерживают только 26-разрядную адресацию памяти, версия v2 включает и 32-разрядные команды умножения. В версии v3 введена полная 32-разрядная адресация для операндов длиной 1 байт и 32-разрядные слова, в версии v4 добавлены полные 64-разрядные команды умножения и команды

загрузки и умножения для 16-разрядных операндов-данных. В версии v5 и ее расширении v5E появились специализированные команды:

- управления точками останова в программах с целью их отладки;
- нормализации чисел для программной реализации арифметических операций с плавающей запятой;
- выполнения операций сложения и умножения над 16-разрядными операндами для программ цифровой обработки сигналов.

На протяжении всего процесса эволюции архитектуры ARM команды всех ее пяти версий кодировались в 32-разрядном формате. Наряду с версиями v4 и v5 существовали их эквиваленты с более компактной кодировкой команд, о которых речь пойдет в следующих разделах.

11.1.1. Система команд Thumb

Помимо полных наборов команд v4 и v5 с 32-разрядной кодировкой спецификация архитектуры системы команд ARM предлагает компактную кодировку подмножества этих наборов. Данное подмножество называется *Thumb*, а соответствующие версии архитектуры ARM носят имена v4T и v5T. Все команды Thumb кодируются 16-разрядным полусловом.

Разработка подсистемы команд Thumb обусловлена необходимостью сократить объем памяти, отводимый для хранения программ, которые управляют недорогими встроенными системами с низким энергопотреблением. На базе архитектуры v4T создан процессор ARM7TDMI. Этот процессор, реализованный в виде единственной микросхемы с небольшим объемом памяти и программным обеспечением для цифровой обработки сигналов, предназначен для использования в мобильных телефонах.

Программы, состоящие из команд Thumb, выполняются следующим образом. Команды извлекаются из памяти и динамически (непосредственно перед выполнением) переводятся из 16-разрядного формата в соответствующий стандартный 32-разрядный формат команд ARM, после чего выполняются. Так они обрабатываются в большинстве недорогих процессоров. В некоторых высокопроизводительных процессорах команды Thumb не преобразуются в 32-разрядный формат, а декодируются из 16-разрядного формата. В регистре текущего состояния программы CPSR (Current Program Status Register) существует разряд T, определяющий, какой формат, Thumb (T = 1) или стандартный 32-разрядный ARM (T = 0), имеет входной поток команд. Допускается чередование в одном приложении команд Thumb и стандартных команд.

Между командами Thumb и стандартными командами ARM есть два существенных различия. Во-первых, во многих командах Thumb используется формат с двумя операндами, где регистр назначения является одним из исходных регистров. Во-вторых, все стандартные инструкции ARM поддерживают механизм условного выполнения команд (по предположению), тогда как в подсистеме Thumb это справедливо только для команд перехода и ряда других команд. Вот почему команды Thumb можно представлять в виде 16-разрядных слов.

11.1.2. Ядра процессоров

Компания ARM разрабатывает и лицензирует спецификации процессоров ARM и тесно связанных с ними компонентов, таких как кэш-память и блоки управления памятью. Эти спецификации приобретаются компаниями-производителями встраиваемых систем и других специализированных компьютерных компонентов. Как правило, схемы процессора ARM интегрируются на одной микросхеме со специализированным аппаратным обеспечением, предназначенным для конкретного типа устройств. Поэтому схемы ARM называют *ядром*. Спецификации компании ARM делятся на две категории: *спецификации аппаратного макроэлемента* и *синтезируемые спецификации*. Первая категория охватывает подробные спецификации физической организации схемы, ориентированные на процесс производства конкретной микросхемы. Ко второй категории относятся программные модули на языке высокого уровня, которые синтезируются из библиотечных компонентов в соответствии с желаемой технологией. Такая спецификация допускает настройку множества разнообразных параметров, определяющих функциональные элементы процессора. Ядро ARM7TDMI процессора разработано в форме спецификации аппаратного макроэлемента, а ядро ARM7TDMI-7 — в форме синтезируемой спецификации.

Компания ARM разработала ядра двух типов: обычные процессорные и CPU. Процессорное ядро содержит только процессор и связанные с ним соединения адресной шины и шины данных. Ядро CPU кроме процессора включает также кэш и блок управления памятью. Название CPU не совсем точное, поскольку обычно оно означает центральное процессорное устройство (ЦПУ). Однако мы употребляем его, так как это термин, посредством которого компания ARM идентифицирует класс своих устройств. Далее приведены краткие описания некоторых типичных процессорных ядер и ядер CPU.

Процессорное ядро ARM7TDMI

Это ядро обычно используется в недорогих устройствах с низким потреблением энергии. Процессор ARM7TDMI включает простой 3-ступенчатый конвейер, состоящий из нескольких ступеней: выборки, декодирования и выполнения. В нем реализована версия v4T архитектуры ARM, поддерживающая как стандартный набор команд, так и команды Thumb. Типичными рабочими параметрами являются напряжение питания 3,3 В и тактовая частота 66 МГц. Возможны и другие варианты реализации, например с напряжением питания 0,9 В для устройств с низковольтными батареями или с тактовой частотой 100 МГц для устройств с более высокой производительностью.

Процессорные ядра ARM9TDMI и ARM10TDMI

Процессорные ядра ARM9TDMI и ARM10TDMI основаны на 5-ступенчатом и 6-ступенчатом конвейерах соответственно. Для достижения более высокой производительности, чем у процессора ARM7TDMI, в них предусмотрены раздельные порты команд и данных. На тактовых частотах 200 и 300 МГц уровни производительности версий 7, 9 и 10 данного подсемейства процессоров ARM соотносятся как 1:2:4. Шина каждого порта памяти процессора ARM10TDMI шире, чем

у двух других процессоров, и равна 64 битам, в то время как у процессоров ARM9TDMI и ARM7TDMI данный параметр составляет только 32 бита. В процессоре ARM9TDMI реализована версия v4T системы команд ARM, а в процессоре ARM10TDMI — версия v5TE. Оба процессора непосредственно декодируют команды Thumb для выполнения. Более высокая производительность может быть достигнута в случае использования с этими процессорами кэш-памяти.

CPU-ядро ARM720T

CPU-ядро ARM720T включает процессорное ядро ARM7TDMI, унифицированный 8-килобайтовый кэш для команд и данных, а также аппаратное обеспечение, назначение которого — управление виртуальной памятью. Включающий 4 канала множественно-ассоциативный кэш состоит из 16-байтовых блоков. В блоке управления памятью применяется 64-элементный ассоциативный буфер быстрого преобразования адреса для хранения адресов страниц памяти, использовавшихся последними. Тактовая частота этого интегрированного устройства может достигать 60 МГц. По сравнению с микросхемой, содержащей только процессорное ядро, встраиваемые кэш-память и блок управления памятью увеличивают общую площадь микросхемы в пять раз, а потребление энергии — втрое.

CPU-ядра ARM920T и ARM1020E

Эти CPU-ядра, основанные на процессорных ядрах ARM9TDMI и ARM10TDMI, снабжены раздельными кэшами команд и данных. Любой кэш ядра ARM920T имеет объем 16 Кбайт и 64-канальную множественно-ассоциативную структуру, а также состоит из 32-байтовых блоков. Для каждого порта памяти выделен отдельный блок управления памятью. Кроме того, все они содержат 64-элементный ассоциативный буфер TLB. В ядре ARM1020E есть два кэша, объемом 32 Кбайт, имеющих такую же структуру, как и кэш ядра ARM9TDMI.

CPU-ядро StrongARM SA-110

CPU-ядро StrongARM — это совместная разработка компаний ARM и Digital Equipment Corporation (последняя в настоящее время входит в состав компании Compaq). Его версия SA-110 производится корпорацией Intel. В этом процессорном компоненте реализована версия v4 архитектуры ARM. Данный процессор не поддерживает набор команд Thumb, но в остальном совместим с процессорным ядром ARM9TDMI. По производительности ядро StrongARM SA-110 близко к ядру ARM920T, но в отличие от последнего оно реализовано по более старой технологии, потребляет больше энергии и работает на тактовой частоте 200 МГц.

Процессор StrongARM содержит 5-ступенчатый конвейер. В нем предусмотрены отдельные кэши команд и данных. Оба кэша имеют 32-канальную множественно-ассоциативную структуру и состоят из 32-байтовых блоков. Для повышения производительности устройств цифровой обработки сигналов процессор снабжен высокоскоростной схемой умножителя с задержкой в три и менее такта.

11.2. Семейства процессоров Motorola 680X0 и ColdFire

С процессором Motorola 68000 вы уже познакомились во второй части главы 3. Теперь мы рассмотрим новые процессоры семейства 680X0 и близкого к нему семейства ColdFire.

Процессор Motorola 68000 разработан в 1979 году. За период с начала 1980-х до начала 1990-х годов на рынке персональных компьютеров появились процессоры 68000, 68020, 68030 и 68040, использовавшиеся в компьютерах Apple. Последний член семейства 680X0 — процессор 68060, выпущенный в середине 1990-х годов. Этот процессор, а также семейство близких к нему процессоров ColdFire рассчитаны на рынок встраиваемых систем.

11.2.1. Процессор 68020

Процессор 68020 значительно превосходит по мощности модель 68000, главным образом за счет ряда важнейших архитектурных усовершенствований. Повышение производительности стало возможным вследствие развития технологии производства СБИС, которые снимают многие конструктивные ограничения для микросхем. Все, что сказано в этом разделе о процессоре 68020, относится также к процессорам 68030 и 68040 (ниже вы найдете более подробную информацию об этих процессорах).

Процессор 68020 имеет внешние соединения для 32-разрядных адресов и 32-рарядных данных. Хотя ширина его шины данных составляет 32 бита, этот процессор эффективно работает с устройствами, пересылающими данные по 8, 16 или 32 бита за раз. Он динамически настраивается на ширину шины данных конкретного устройства, причем совершенно прозрачно для программиста. Рассматриваемый процессор содержит линии управления, по которым устройство передает информацию о количестве разрядов пересылаемых данных. Таким образом, процессор способен взаимодействовать с устройствами, которые имеют шины различной ширины, не зная этого параметра до момента, когда устройство инициирует передачу данных.

Для процессора 68000 является обязательным выравнивание операндов длиной в одно слово по четным границам адресов. В процессоре 68020 это ограничение снято, и операнды всех размеров могут располагаться по любым адресам. Вполне допустимо, что 16- и 32-разрядные операнды могут располагаться в смежных 32-битовых областях основной памяти. Для пересылки каждого такого операнда требуется два цикла обращений к памяти, что отражается на производительности процессора. Обращения процессор выполняет автоматически. По адресу он определяет, к каким 32-битовым областям памяти нужно обратиться и в каком порядке соединить отдельные байты, чтобы получить значение операнда.

Набор регистров и типы данных

Подобно процессору 68000, процессор 68020 может функционировать как в пользовательском режиме, так и в режиме супервизора. В пользовательском режиме

программам доступны все регистры, которые были перечислены на рис. 3.18 для процессора 68000. В режиме супервизора у процессора 68020 появляется несколько дополнительных управляющих регистров, назначение которых состоит в том, чтобы упростить реализацию программного обеспечения операционной системы.

Адресуемыми единицами данных процессора 68000 являются бит, байт, слово, длинное слово и упакованное двоично-десятичное число (BCD). Наряду с ними процессор 68020 может адресовать четверное слово, неупакованное слово BCD и битовое поле. Размер четверного слова составляет 64 бита. Неупакованные числа в формате BCD занимают по одному байту на цифру BCD. Битовое поле состоит из переменного количества битов 32-разрядного длинного слова и определяется положением крайнего слева бита, а также количеством битов в поле.

Режимы адресации

Все режимы адресации процессора 68000, перечисленные в табл. 3.2, поддерживаются и процессором 68020. Более того, для процессора 68020 разработано несколько дополнительных версий индексного режима, которые обеспечивают большую гибкость при доступе к списочным структурам данных и адресов.

Полный индексный режим стал эффективнее, поскольку в нем предусмотрена поддержка широкого диапазона смещений и коэффициента масштабирования. Синтаксис процессора 68000 для полного индексного режима таков:

disp(A*n*,R*k*.size)

Здесь *disp* — это смещение со знаком, а *size* служит для указания количества разрядов регистра R*k* (32 или 16), используемых при вычислении исполнительного адреса. В версии 68020 допустимо 8-, 16- или 32-разрядное смещение. Поддерживается коэффициент масштабирования, на который умножается содержимое регистра R*k*. Он может быть равным 1, 2, 4 или 8. Ознакомимся с синтаксисом этого режима:

(disp,A*n*,R*k*.size*scale)

Обратите внимание на то, что смещение задано в скобках. Исполнительный адрес EA вычисляется следующим образом:

$$EA = disp + [An] + ([Rk] \times scale)$$

Этот режим полезен в процессе работы со списками элементов длиной 1, 2, 4 или 8 байт. Если коэффициент масштабирования равен размеру элемента, для доступа к последовательным элементам списка достаточно каждый раз увеличивать значение R*k* на 1.

Еще одним эффективным способом индексной адресации является косвенный индексный режим с доступом через память, при котором операнд, содержащий адрес, неявно считывается из основной памяти. Существуют две разновидности этого режима. В *постиндексном косвенном режиме с доступом через память* адрес извлекается из памяти перед индексацией. Синтаксис данного режима приведен ниже.

([basedisp,A*n*],R*k*.size*scale,outdisp)

Исполнительный адрес для этого режима вычисляется так:

$$EA = [basedisp + [An]] + ([Rk] \times scale) + outdisp$$

Интересно то, что смещение используется дважды. *Базовое* смещение, величина которого составляет 16 или 32 бита, необходимо для преобразования адреса в адресном регистре A*n*, который затем применяется с целью выборки операнда-адреса из памяти. Это позволяет извлечь адрес из списка адресов, хранящегося в памяти начиная с адреса, заданного в A*n*. *Внешнее* смещение производится во время индексной адресации.

В *преиндексном косвенном режиме с доступом через память* основная часть процедуры индексации осуществляется перед выборкой адреса. Ознакомимся с синтаксисом этого режима:

([basedisp,A*n*,R*k*.size*scale],outdisp)

Исполнительный адрес вычисляется следующим образом:

$$EA = [basedisp + [An] + ([Rk] \times scale)] + outdisp$$

В обоих режимах значения A*n*, R*k*, *basedisp* и *outdisp* необязательны. Если они не заданы, исполнительный адрес вычисляется без их учета. Рассмотренные режимы адресации полезны при работе со списками, в которых хранятся адреса элементов данных, а не сами данные. Данные при этом могут находиться в разных местах памяти.

Кроме того, поддерживаются относительные версии всех режимов адресации, в которых вместо адресного регистра A*n* используется счетчик команд.

Набор команд

Процессор 68020 поддерживает все команды процессора 68000, причем некоторые из них проявляют большую гибкость. Например, при использовании команды перехода можно задавать 32-разрядное смещение, а ряд других команд позволяет использовать более длинные операнды. Появилось несколько новых команд, в частности команды для работы с операндами типа битовых полей.

Интегрированный кэш

Микросхема 68020 содержит кэш команд объемом 256 байт, состоящий из 46-разрядных длинных слов. Загрузка новых слов в этот кэш производится по схеме прямого отображения.

11.2.2. Процессоры 68030 и 68040

Процессор 68030 имеет два важных отличия от процессора 68020. Наряду с кэшем команд в нем есть еще один кэш такого же размера, предназначенный для данных. Кэш данных состоит из 16 блоков по 4 длинных слова в каждом. Кроме того, в состав процессора 68030 входит блок управления памятью.

Блок выполнения процессора 68030 генерирует виртуальные адреса. На основе виртуального адреса схема доступа к кэшу определяет, имеется ли в нем нужный

операнд. Параллельно с обращением к кэшу блок управления памятью транслирует виртуальный адрес в физический, который в случае промаха можно без промедлений использовать для доступа к основной памяти.

Процессор 68040 включает блок операций с плавающей запятой, который соответствует стандарту IEEE, описанному в главе 6. Как и процессор 68030, он снабжен кэшами команд и данных, но его блок управления памятью более усовершенствован. В процессоре 68040 имеются два независимых кэша для трансляции адресов, которые позволяют одновременно транслировать адреса команд и данных. Процессор 68040 имеет конвейерную структуру, благодаря которой возможен выбор очередных команд еще до окончания выполнения предыдущих. За пересылку команд и данных из кэшей ответственны две внутренние шины. Вместе с двумя схемами трансляции адресов эти шины обеспечивают одновременный доступ к кэшам команд и данных.

С добавлением в процессор 68040 схем, предназначенных для мониторинга операций на внешней шине, открылись новые возможности по его использованию в мультипроцессорных системах. Одним из основных требований к таким системам является согласование общих данных, которые временно могут располагаться в нескольких кэшах разных процессоров. Как рассказывается в главе 12, схемы мониторинга шины выявляют операции пересылки данных по шине, приводящие к изменению кэшируемых данных.

11.2.3. Процессор 68060

Процессор 68060, последний представитель семейства 680X0, разработан в середине 1990-х годов и предназначен для рынка встраиваемых систем. Он функционирует на тактовых частотах от 50 до 75 МГц и благодаря новой организации и новой технологии производства обладает в 2,5 раза большей производительностью, чем процессор 68040 с тактовой частотой 40 МГц.

Процессор 68060 имеет конвейерную суперскалярную архитектуру. Его конвейер включает четыре базовые ступени, а в случае обратной записи в память используются еще две ступени. На одном такте может начаться выполнение не более трех команд. Основное аппаратное обеспечение блока обработки команд составляют три функциональных арифметических устройства: одно целочисленное и два с плавающей запятой. Существуют интегрированные кэши объемом 8 Кбайт для команд и данных. Каждый кэш имеет четырехканальную множественно-ассоциативную структуру и состоит из 16-байтовых блоков. Кроме того, есть два буфера быстрого преобразования адресов, предназначенных для трансляции виртуальных адресов в физические. Эти буферы содержат по 64 записи и имеют 4-канальную множественно-ассоциативную структуру. С целью увеличения скорости прохождения команд через конвейер используется технология предсказания ветвлений.

11.2.4. Семейство процессоров ColdFire

Начиная с середины 1990-х годов компания Motorola представила на рынке встраиваемых систем целую серию процессорных компонентов и аппаратное

обеспечение для малых компьютеров под названием ColdFire. В основу их разработки положено процессорное ядро 68060. Разнообразные средства, составляющие семейство ColdFire, имеют память небольшого объема, а также последовательные и параллельные порты ввода-вывода. Они различаются потребляемой мощностью и производительностью и адресованы широкому диапазону устройств. В семейство входят также аппаратные спецификации процессоров и синтезируемые программные спецификации.

11.3. Семейство процессоров Intel IA-32

Процессоры Intel пользуются большим успехом и широко применяются в ноутбуках и персональных компьютерах. В 1980-х годах фирма Intel выпустила первую серию процессоров для использования в IBM PC. Они базировались на процессоре 8086, созданном в 1979 году, который генерировал на внешней шине 20-битовые адреса, а также оперировал 16-разрядными данными. (В первом IBM PC использовалась недорогая 8-разрядная версия процессора 8086, называвшаяся 8088.) Поскольку процессор 8086 был заключен в 40-контактный корпус, адреса и данные мультиплексировались на одни и те же контакты.

Архитектура процессоров Intel постоянно совершенствовалась, на основе базовой системы команд выпускались все более мощные процессоры. Эволюционный ряд включает процессоры 80286, 80386, 80486, а также процессоры текущей серии Pentium. Процессор 80286 был 16-разрядным, последующие процессоры оперировали 32-разрядными адресами и данными. Первым представителем семейства IA-32 стал процессор 80386. Благодаря тому что 32-разрядные микросхемы выпускаются в виде модулей большего размера, исчезла необходимость в мультиплексировании линий адреса и данных.

11.3.1. Сегментация памяти для семейства процессоров IA-32

В разделе 3.16.1 кратко рассказывалось об использовании сегментных регистров в архитектуре IA-32 для формирования адресов памяти (см. рис. 3.37). Мы продолжим развивать эту тему. Для начала читателю полезно будет узнать, как сегментные регистры использовались в процессоре 8086. Более современные процессоры IA-32 способны функционировать в так называемом *реальном* режиме, где они могут выполнять машинный программный код процессора 8086.

Реальный режим

Реальный режим процессоров IA-32 — это режим генерирования адресов, используемый процессором 8086. В нем память рассматривается как последовательность сегментов объемом по 64 Кбайт. Для каждого сегмента 16-разрядные исполнительные адреса формируются с использованием режимов адресации процессора 8086. Для доступа к сегментам кода, стека и двум сегментам данных в этом процессоре применяются сегментные регистры CS, SS, DS и ES. В процессор 80386 были включены еще два сегментных регистра — FS и GS.

На рис. 11.1 представлен процесс генерирования 20-разрядных внешних адресов памяти. Заданное в сегментном регистре 16-разрядное значение сдвигается на четыре позиции влево, вследствие чего получается 20-разрядный адрес памяти, представляющий собой начальный адрес сегмента. Для получения 2-разрядного исполнительного адреса к начальному адресу сегмента добавляется сгенерированный процессором 16-разрядный исполнительный адрес, обозначенный на рисунке как смещение.

Чтобы локализовать в памяти сегмент, в сегментный регистр помещаются 16 старших разрядов 20-разрядного адреса начала этого сегмента. В адресном пространстве, занимающем 1 Мбайт и покрываемом 20-разрядными адресами, может располагаться 16 неперекрывающихся сегментов объемом 64 Кбайт. Однако сегменты могут и перекрываться. Это удобно при организации совместного доступа к командам и данным разными программами. Сегменты CS и SS используются в тех случаях, когда в программе имеются ссылки на команды или стек. Для хранения данных по умолчанию выделяется сегментный регистр DS. Если же для доступа к данным команды должен применяться регистр ES, в начало этой команды добавляется соответствующий код.

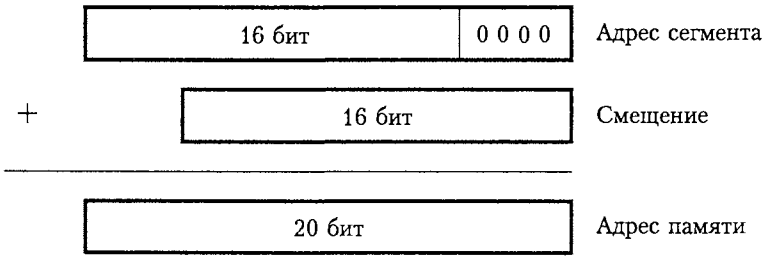


Рис. 11.1. Формирование адреса памяти в процессоре 8086

Защищенный режим

Процессоры архитектуры IA-32 обычно работают в *защищенном* режиме. На рис. 11.2 показано, как в этом режиме генерируется физический адрес памяти на основе содержимого базового и индексного регистров, а также содержащегося в команде значения смещения. Для получения 32-разрядного исполнительного адреса значение индексного регистра умножается на коэффициент масштабирования, равный 1, 2, 4 или 8, затем результат прибавляется к содержимому базового регистра с учетом заданного в команде смещения. Четырнадцать старших битов одного из шести сегментных регистров (показанных на рис. 3.37) определяют *дескриптор*, используемый в качестве индекса в таблице дескрипторов сегментов, из которой извлекается 32-разрядный базовый адрес. Этот адрес прибавляется к исполнительному адресу внутри сегмента, вследствие чего получается 32-разрядный *линейный адрес*. Страничный блок, используя таблицу страниц, транслирует линейный адрес в 32-разрядный физический адрес.

Таблицы страниц и дескрипторов сегментов довольно велики, поэтому они хранятся в основной памяти. Для обеспечения быстрой трансляции адресов может использоваться описанный в главе 5 буфер быстрого преобразования адресов

TLB. В таблицах дескрипторов сегментов содержатся поля прав доступа, а также поля границ сегментов, определяющие их максимальный размер. Этими параметрами управляет операционная система. Они нужны для защиты как операционной системы, так и прикладных программ, находящихся в основной памяти. Отсюда и название данного режима работы процессора — «защищенный».

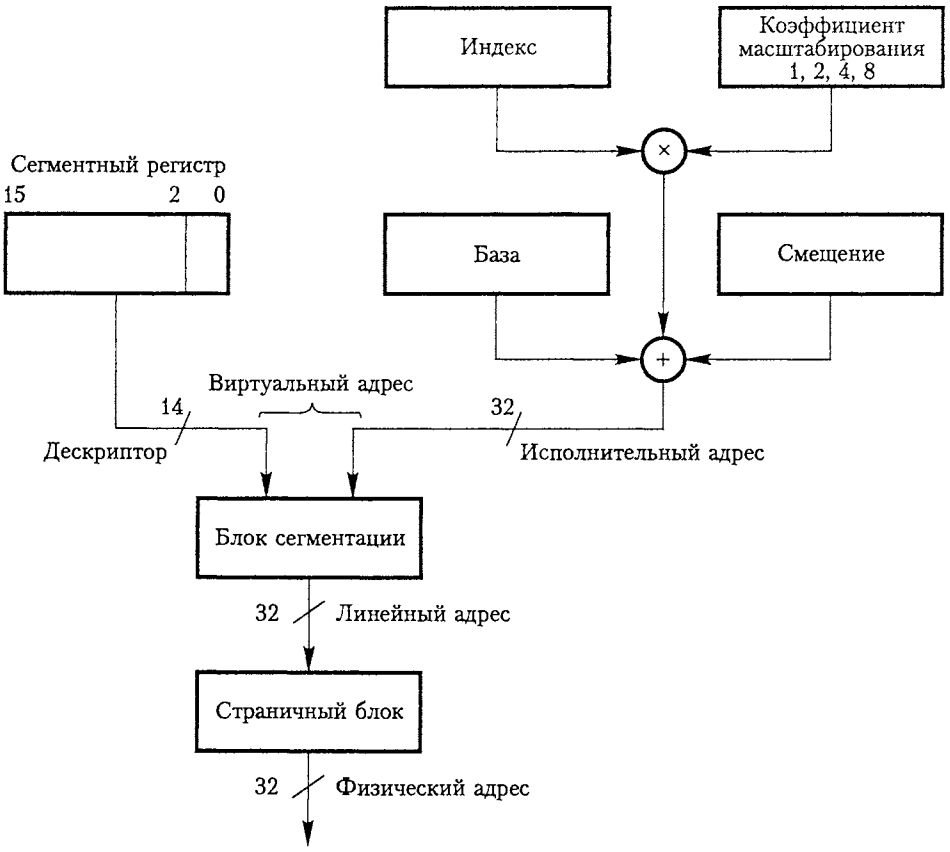


Рис. 11.2. Формирование адреса в архитектуре IA-32

Сегменты и страницы используются для организации памяти одним из следующих способов:

- в виде плоского адресного пространства, в котором исполнительный адрес играет роль физического;
- в виде одного или нескольких сегментов переменой длины (без разбиения на страницы);
- в виде 32-разрядного пространства памяти, разделенного на страницы объемом 4 Кбайт;
- в виде структуры, состоящей из сегментов и страниц.

11.3.2. 16-разрядный режим

Процессоры IA-32 могут функционировать в режиме, допускающем непосредственное выполнение программ на машинных языках ранних 16-разрядных процессоров Intel (8086 и 80286). В этом режиме применяются только младшие половины регистров процессора, обозначенные на рис. 3.38 как AX, CX,.... Размер внутренних адресов достигает 16 разрядов, а для адресации используется только подмножество режимов, перечисленных в табл. 3.3. Так, значение индексного регистра при формировании исполнительного адреса масштабировать невозможно.

Переключение между 32-разрядным и 16-разрядным режимами может выполняться на уровне отдельных команд. Кроме того, возможен независимый выбор разрядности адреса и данных. Поэтому при работе с 16-разрядными данными могут использоваться все способы адресации архитектуры IA-32. Перед началом выполнения программы устанавливается режим, используемый по умолчанию. Для переключения в другой режим к команде добавляется префиксный байт, не показанный на рис. 3.41.

11.3.3. Процессоры 80386 и 80486

В процессоре 80386, как уже было сказано, впервые реализована архитектура IA-32, описанная в главе 3. Он поддерживает технологии сегментации памяти и разбиения ее на страницы, с которыми мы познакомились в разделе 11.3.1.

Процессор 80486 относится к категории первых микросхем, содержащих порядка 1 млн. транзисторов — приблизительно столько же, сколько их насчитывается в процессоре Motorola 68040. Благодаря расширенным схемам по производительности он значительно превосходил процессор 80386. Арифметическое устройство с плавающей запятой соответствовало стандарту IEEE (см. главу 6). В компьютерах на базе процессоров 80386 за выполнение вычислений с плавающей запятой отвечала отдельная микросхема сопроцессора. Поддержка страниц и управление памятью в процессоре 80486 осуществлялись так же, как и в процессоре 80386.

Процессор 80486 содержал 4-канальный множественно-ассоциативный кэш для команд и данных. Загрузка в него новой информации ускорялась благодаря механизму пакетной пересылки данных, позволявшему считывать и загружать в кэш четыре 32-разрядных слова одним блоком. В нем применялся протокол со сквозной записью, согласно которому записываемые в кэш данные сразу автоматически сохранялись и в основной памяти.

С целью повышения производительности в архитектуру процессора 80486 был включен ряд устройств параллельной и конвейерной обработки команд. Устройства обработки целых чисел и чисел с плавающей запятой могли выполнять команды параллельно. Во время обработки одной команды из памяти уже извлекалась следующая команда. Для выполнения наиболее «популярных» команд требовалось меньше тактов, чем в процессоре 80386. Так, команды загрузки и сохранения данных, а также команды, производящие операции над данными регистров без обращения к памяти, выполнялись за один такт.

11.3.4. Процессор Pentium

У процессора Pentium, выпущенного в 1993 году, производительность значительно выше, чем у процессора 80486. Он содержит 3 млн. транзисторов и при выполнении программ, осуществляющих целочисленные вычисления, имеет вдвое высшую вычислительную мощность, чем процессор 80486. В процессе выполнения программ, предназначенных для интенсивных вычислений с плавающей запятой, данный показатель возрастает в пять раз.

Pentium имеет CISC-архитектуру, но при этом для повышения производительности в нем использованы многие структурные элементы RISC-процессоров. (Они применялись и в процессоре 80486, но в меньшем количестве.) В частности, для команд и данных предусмотрены раздельные кэши объемом по 8 Кбайт, интегрированные в микросхему. У процессора Pentium суперскалярная организация. Он содержит несколько конвейерных блоков, загружающих по две команды за такт. Благодаря 64-разрядной внешней шине данных ускоряется загрузка в кэши данных из внешней памяти. Кэши имеют 2-канальную множественно-ассоциативную структуру и состоят из 32-байтовых блоков. В состав процессора входят три независимых конвейерных операционных блока: два для целочисленных операций и один для операций с плавающей запятой. Целочисленные конвейеры имеют пять ступеней, а устройство с плавающей запятой — восемь ступеней.

В процессоре Pentium используется простая форма динамического предсказания переходов. Каждый раз выбирается то направление, по которому осуществлялся предыдущий переход. Такое предсказание верно для всех переходов, выполняемых в конце программного цикла после первого перехода и до выхода из цикла.

11.3.5. Процессор Pentium Pro

Процессор Pentium Pro с тактовой частотой 133 МГц выпущен в 1995 году. Его производительность вдвое выше, чем у процессора Pentium с тактовой частотой 100 МГц. Такой скорости удалось достичь за счет суперскалярного и внеочередного выполнения команд. Коэффициент суперскалярного выполнения, соответствующий количеству команд, которые загружаются за один такт, у процессора Pentium Pro равен 3, а у процессора Pentium — 2. Его конвейеры содержат по 12 ступеней, тогда как в целочисленных конвейерах Pentium их насчитывается 5. Ширина шин данных внутри процессора составляет 64 разряда, что вдвое больше, чем у процессора Pentium. В процессоре Pentium Pro, как и в Pentium, предусмотрены раздельные, интегрированные в микросхему кэши первого уровня (L1) для команд и данных объемом 8 Кбайт каждый. Кроме того, добавлен кэш второго уровня (L2) емкостью 256 Кбайт. Он располагается в одном модуле с микросхемой процессора, но на отдельной микросхеме, соединенной с микросхемой процессора 64-разрядной шиной.

Суперскалярная обработка обеспечивается наличием нескольких блоков выполнения команд — двумя для целочисленных операций и двумя для операций с плавающей запятой. Одним из важнейших усовершенствований, внесенных в процессор Pentium Pro и обеспечивших столь значительное повышение его производительности, является возможность активизации команд не в том порядке,

в каком они заданы в программе и выбраны из памяти. Эта функция позволяет параллельно выполнять большее количество команд. Конечно, для получения правильных результатов вычислений требуются дополнительные управляющие схемы. Как и в процессоре Pentium, в Pentium Pro происходит динамическое предсказание переходов, цель которого — заблаговременный выбор команд для параллельной работы.

Внешние схемы мониторинга шины позволяют использовать процессор Pentium Pro в мультипроцессорных системах. Эти схемы служат для приведение в соответствие общих данных, временно хранящихся в кэш-памяти нескольких процессоров. (Подробнее о согласовании данных в кэш-памяти рассказывается в главе 12.)

11.3.6. Процессоры Pentium II и Pentium III

В архитектуру системы команд процессора Pentium II добавлены команды MMX, о которых кратко рассказывалось в разделе 3.23.2. Эти команды обеспечивают параллельную обработку небольших групп чисел, представляющих мультимедиаданные (пикселы, оцифрованный звук и т. п.). Для обработки такого рода данных используются те же восемь 64-разрядных регистров, которые применяются в случае данных с плавающей запятой. Кэши L1 в процессоре Pentium II имеют вдвое больший размер, чем в процессоре Pentium Pro, то есть по 16 Кбайт каждый. Размер внешнего кэша L2 составляет 512 Кбайт.

В процессор Pentium III добавлена поддержка векторных команд (SIMD), описанных в разделе 3.23.3. Эти команды, называемые потоковым расширением SIMD (SSE), обеспечивают эффективное выполнение векторных операций над данными с плавающей запятой. В каждый из восьми новых 128-разрядных регистров, называемых регистрами XMM, упаковывается по четыре 32-разрядных операнда с плавающей запятой. Кэши процессора Pentium III такие же, как у Pentium II, с единственным, но очень важным отличием: кэш-память второго уровня емкостью 256 Кбайт расположена на той же микросхеме, что и процессор, благодаря чему увеличена ширина его соединения с кэшами первого уровня.

Процессоры Pentium, Pentium Pro, Pentium II и Pentium III при выпуске в 1993, 1995, 1997 и 1999 годах имели тактовую частоту 60, 200, 266 и 500 МГц соответственно. В дальнейшем благодаря непрерывному усовершенствованию технологий производства СБИС, заключающемуся в постоянном уменьшении размеров транзисторов и вентильных задержек, тактовая частота версий процессора Pentium III достигла 1 ГГц.

11.3.7. Процессор Pentium 4

Процессор Pentium 4 выпущен в 2000 году и имеет тактовую частоту от 1,3 до 1,5 ГГц. Он поддерживает весь набор команд IA-32, включая команды MMX и SSE. Расширенный набор команд SSE (SSE2) позволяет обрабатывать по два упакованных 64-разрядных числа с плавающей запятой или по два упакованных 64-разрядных целых числа, хранящихся в 12-разрядных регистрах XMM. Такие

длинные целые числа удобны для шифрования и дешифрирования — операций, выполняемых в приложениях с функциями защиты данных. Повышение тактовой частоты достигается в значительной мере за счет использования длинных конвейеров с короткими ступенями (теперь их количество увеличено вдвое по сравнению с Pentium III, где их было всего 10), а также за счет усовершенствованных структуры и технологий производства схем.

Процессор содержит раздельные кэши первого уровня для команд и данных. Кэш данных имеет объем 8 Кбайт, 4-канальную множественно-ассоциативную структуру и состоит из блоков по 64 байта. Кэш команд предназначен для хранения декодированных сегментов потока команд, которые могут включать одну или несколько ветвей исходной программы. В случае повторения сегментов выполнение программы протекает быстрее, но требуется проверка, действительно ли каждый раз осуществляется переход к одной и той же ветви программы. Эта стратегия называется *кэшированием с отслеживанием* (trace cache). Декодированные команды представлены как микрооперации. Каждая команда IA-32 способна определять до восьми микроопераций. В кэше с отслеживанием может находиться множество сегментов потока команд, содержащих до 12 тысяч микроопераций.

Интегрированный в микросхему кэш второго уровня объемом 256 Кбайт состоит из 128-байтовых блоков и обладает 8-канальной множественно-ассоциативной структурой. Соединение между кэшами L1 и L2 обеспечивает пересылку данных со скоростью 48 Гбайт/с, тогда как в процессоре Pentium III этот параметр равен 16 Гбайт/с.

Системная шина процессора Pentium 4 тоже гораздо быстрее шины процессора Pentium III. Она имеет ширину 64 разряда и работает на частоте 400 МГц. Скорость пересылки данных по этой шине составляет 3,2 Гбайт/с, а у процессора Pentium III — 1 Гбайт/с.

11.3.8. Процессоры Advanced Micro Devices IA-32

Процессоры на базе архитектуры IA-32 производятся и другими компаниями, конкурирующими с Intel, в частности фирмой Advanced Micro Devices (AMD). В 2000 году AMD выпустила процессор Athlon, работающий на тактовой частоте 1,2 ГГц, то есть не уступающий по данному показателю процессору Intel Pentium 4.

Athlon — это суперскалярный процессор, у котого кэши обоих уровней расположены на микросхеме процессора. Кэш первого уровня, разделяемый на кэш команд и кэш инструкций имеет общую емкость 128 Кбайт, а кэш второго уровня — емкость 256 Кбайт. Интерфейс между процессором и основной памятью типа DDR DRAM (см. главу 5) обеспечивает максимальную скорость пересылки данных 2,1 Гбайт/с. Протоколы системной шины ввода-вывода обеспечивают работу этой шины на частоте 200 или 266 МГц.

11.4. Семейство процессоров PowerPC

В начале 1990-х годов компании IBM, Motorola и Apple совместными усилиями разработали семейство процессоров RISC-типа для персональных компьютеров и рабочих станций, которое получило название PowerPC. Процессоры PowerPC,

производимые компаниями IBM и Motorola, использовались в компьютерах IBM и Apple. По вычислительной мощности эти процессоры сравнимы с процессорами Intel IA-32, которые выпускались в то же время. Первым процессором архитектуры PowerPC был процессор 601, увидевший свет в 1993 году. В настоящем разделе мы рассмотрим архитектуру семейства PowerPC, а затем поговорим о нескольких его представителях.

11.4.1. Набор регистров

В архитектуре PowerPC определено 32 регистра общего назначения и 32 регистра для данных с плавающей запятой. Регистры с плавающей запятой имеют длину 64 бита. Представление чисел с плавающей запятой соответствует стандарту IEEE. В архитектуре PowerPC определены как 32-разрядный, так и 64-разрядный режимы функционирования. Размер регистров общего назначения определяется тем, какой из режимов реализован в конкретном процессоре.

11.4.2. Режимы адресации памяти

Память адресуется побайтово, доступ к ней производится с помощью команд загрузки и сохранения, осуществляющих пересылку операндов-данных из памяти в регистры и в обратном направлении. Согласно концепции RISC, используются только простые формы индексной адресации. Исполнительный адрес генерируется путем прибавления значения базового регистра к индексу, представленному либо непосредственно заданным в команде значением, либо содержимым индексного регистра. При желании исполнительный адрес может быть загружен обратно в базовый регистр для упрощения операции считывания или сохранения набора операндов по последовательным адресам памяти. Имеются специальные команды для одновременной пересылки нескольких операндов. У команд загрузки и сохранения существует множество версий, что обеспечивает большую гибкость при пересылке операндов различных типов и размеров.

11.4.3. Команды

Команды PowerPC имеют длину 32 разряда и единообразный формат. По отношению к арифметическим и логическим командам применяется 3-регистровый формат, в котором задаются два регистра-операнда и регистр назначения, используемый для сохранения результата. Поддерживается большое количество команд перехода. Заслуживает внимания команда MultiplyAdd, производящая операцию

$$RD \leftarrow ([RA] \times [RB]) \pm [RC]$$

над операндами с плавающей запятой, которые хранятся в регистрах RA, RB и RC. Отдельный класс команд перехода уменьшает значение счетчика, а затем выполняет переход в зависимости от того, равно ли результирующее значение счетчика нулю. Команда MultiplyAdd, команды декремента-перехода и пересылки нескольких операндов от регистра к регистру, а также операция необязательного

обновления базового регистра при индексной адресации не типичны для архитектуры RISC. Они полезны в первую очередь для выполнения арифметических операций в приложениях обработки сигналов, эффективного завершения циклов, сохранения и восстановления регистров процессора на входе и выходе подпрограмм, а также обработки списков элементов данных. Однако при проектировании не учитывалось их влияние на эффективность прохождения потока команд через конвейер процессора. Все эти команды, за исключением команды декремента-перехода, присутствуют и в системе команд процессоров ARM.

11.4.4. Процессоры PowerPC

Архитектура PowerPC является развитием архитектуры POWER, использовавшейся в процессорах компьютеров IBM Risc System (RS)/6000. Первой реализацией архитектуры PowerPC был процессор 601, ставший связующим звеном между двумя архитектурами. В этом процессоре было внедрено надмножество команд POWER и PowerPC, что позволило ему выполнять как откомпилированные программы на машинном языке процессора POWER, так и программы PowerPC. Последующие процессоры данного семейства были уже «чистыми» процессорами PowerPC.

Процессор PowerPC 601

Процессорная микросхема 601, содержащая 2,8 млн транзисторов, изначально использовалась в настольных компьютерах IBM. Это 32-разрядный процессор, предназначенный для ноутбуков, настольных компьютеров и недорогих мультипроцессорных систем. Были выпущены его разные версии с тактовыми частотами 50, 66, 80 и 100 МГц.

Процессор PowerPC 601 и для команд, и для данных имеет 32-килобайтовый кэш на микросхеме процессора. Он обладает 4-канальной множественно-ассоциативной структурой. Процессор содержит три независимых исполнительных блока: целочисленный, с плавающей запятой и блок обработки переходов. За один такт в исполнительные блоки процессора может загружаться до трех команд. У целочисленного конвейера насчитывается 4 ступени, а у конвейера с плавающей запятой — 6 ступеней.

Процессор PowerPC 603

Процессор 603 также относится к числу 32-разрядных. Он рассчитан на использование в ноутбуках и настольных компьютерах, имеет невысокую стоимость и малую мощность. На частоте 80 МГц данный процессор потребляет около 3 Вт. Пять входящих в его состав операционных блоков способны работать параллельно, как и аппаратное обеспечение, ответственное за подготовку и выполнение команд, которое может выдавать до трех команд за такт и имеет более сложную структуру по сравнению с процессором 601. Расположенный на микросхеме процессора кэш разделен на два отдельных кэша по 8 Кбайт для временного хранения команд и данных.

Процессор PowerPC 604

Для систем с более высокой производительностью, чем у процессоров 601 и 603, разработан 32-разрядный процессор 604. Скорость целочисленных вычислений

и вычислений с плавающей запятой в нем вдвое выше, чем у двух указанных моделей. Такой высокий уровень производительности достигается за счет тактовой частоты 100 МГц и суперскалярной организации, позволяющей загружать до четырех команд за такт. В состав процессора входят шесть независимых операционных блоков: три целочисленных блока, блок с плавающей запятой, блок загрузки/сохранения и блок обработки переходов. Процессор PowerPC 604 рассчитан на рынок персональных компьютеров и рабочих станций среднего класса.

Процессор PowerPC 620

Процессор 620 реализует полную 64-разрядную архитектуру PowerPC и поддерживает суперскалярное выполнение команд. Он предназначен для высокопроизводительных настольных компьютеров, серверов, систем обработки транзакций и мультипроцессорных систем.

Как и процессор 604, PowerPC 620 содержит шесть независимых операционных блоков и способен загружать до четырех команд за один такт. Скорость выполнения команд в конкретной программе повышается за счет их внеочередного выполнения. В процессоре используется технология предсказания переходов. Микросхема процессора содержит раздельные кэши для команд и данных объемом 32 Кбайт каждый. Оба кэша имеют 8-канальную множественно-ассоциативную структуру.

Процессор MPC7450

Процессоры 601, 603, 604 и 620 — это первые процессоры PowerPC производства IBM. После линейки моделей 6XX компания Motorola создала линейки моделей 7XX и 7XXX, в обозначения которых был добавлен префикс MPC. Последним процессором линейки MPC7XXX стал процессор MPC7450 с тактовой частотой 733 МГц, выпущенный в начале 2001 года и применяемый в компьютерах Apple Power Mac G4.

MPC7450 — это суперскалярный процессор с 7-ступенчатым конвейером. В его функциональные блоки может загружаться до четырех команд за такт. Всего насчитывается 11 таких блоков: блок загрузки/сохранения, блок обработки переходов, четыре целочисленных блока, блок операций с плавающей запятой и четыре блока, выполняющих параллельные арифметические операции с упакованными векторными операндами-данными. Блоки, принадлежащие к последней категории, согласно терминологии Motorola, называются AltiVec.

Аппаратное обеспечение AltiVec производит параллельные операции с упакованными векторными операндами-данными подобно тому, как процессоры Intel Pentium выполняют операции MMX и SSE, описанные в разделах 3.23.2 и 11.3.6. Упакованные данные, обрабатываемые командами AltiVec, располагаются в тридцати двух 128-разрядных векторных регистрах, которые отделены от регистров общего назначения и регистров с плавающей запятой. В векторных регистрах может храниться шестнадцать 8-битовых целых чисел, восемь 16-битовых целых чисел, четыре 32-битовых целых числа или четыре числа с плавающей запятой одинарной точности (32 бита). Для обмена данными между памятью и векторными регистрами используются команды загрузки и сохранения векторов. Команды AltiVec ускоряют работу мультимедийных приложений и приложений обработки сигналов. Одной из них является команда Multiply-Accomulate, перемножающая

необходимые элементы двух векторов и прибавляющая произведения к соответствующим элементам третьего векторного регистра. Эта операция применяется для цифровой обработки сигналов. Кроме того, имеются команды для вычисления точечного произведения векторов.

Кэш-память первого уровня размещается на микросхеме процессора и состоит из двух отдельных 32-килобайтовых кэшей команд и даных 8-канальной множественно-асоциативной стуктуры. Кэш-память второго уровня тоже находится на микросхеме процессора, имеет объем 256 Кбайт и 8-канальную множественно-асоциативную стуктуру. Обмен данными между кэшами L1 и L2 осуществляется по 256-разрядному соединению на тактовой частоте процессора. Имеется еще и кэш L3, связанный с процессором 64-разрядной шиной. Его емкость может составлять 1 или 2 Мбайт.

11.5. Семейство процессоров SPARC компании Sun Microsystems

SPARC — это масштабируемая архитектура, разработанная компанией Sun Microsystems Corporation и лежащая в основе серии высокопроизводительных процессоров. Все эти процессоры характеризуются однотипной базовой архитектурой системы команд и предназначены для рынка высокопроизводительных рабочих станций и серверов. Системы данного типа описаны в главе 12.

Архитектура SPARC относится к типу RISC. Она использует 3-регистровые 32-разрядные команды фиксированной длины. В команде задаются два исходных регистра-операнда и один регистр назначения. Все команды, выполняющие операции с данными, применяют только регистры процессора. Существует 32 регистра общего назначения для хранения целых чисел и адресов, а также 32 регистра для операндов с плавающей запятой. Полномочиями доступа к памяти обладают только команды загрузки и сохранения, осуществляющие пересылку операндов из регистров в основную память и в обратном направлении.

Первые реализации архитектуры SPARC, увидевшие свет в 1987 году, обрабатывали 32-разрядные адреса и 32-разрядные данные. В последней ее версии, имеющей номер 9, длина адресов и данных достигает 64 разрядов. Указанная версия реализована в серии процессоров UltraSPARC. Команды остаются 32-разрядными, а регистровая модель, которой пользуются программисты, — неизменной. Поддерживается обратная совместимость, то есть процессоры UltraSPARC способны выполнять код, созданный для ранних версий архитектуры.

С архитектурой SPARC вы познакомились в главе 8, где на примере процессора UltraSPARC II демонстрировалась реализация конвейерной обработки команд в высокопроизводительных процессорах. Семейство UltraSPARC, включающее процессоры UltraSPARC I, II и III, характеризуется большим количеством операционных блоков и суперскалярной организацией. Наряду с базовым набором команд SPARC в нем определено множество специализированных команд для поддержки графических и мультимедийных приложений, которые образуют *набор визуальных команд* (Visual Instruction Set, VIS). Команды VIS подобны командам

Intel MMX и SSE, а также командам Motorola AltiVec. Все процессоры семейства UltraSPARC характеризуются повышенной производительностью, более быстродействующими памятью и интерфейсами ввода-вывода, а также более объемной и сложной кэш-памятью. Так, процессор UltraSPARC I производится по 0,5-микронной КМОП-технологии и работает на тактовой частоте 167 МГц. Следующий за ним процессор UltraSPARC II имеет очень похожую 9-ступенчатую конвейерную организацию, но благодаря 0,25-микронной технологии обладает более высокой производительностью. Он работает на тактовых частотах от 250 до 480 МГц.

Последний член рассматриваемого семейства, процессор UltraSPARC III, включает 14-ступенчатый конвейер. Он содержит четыре целочисленных операционных блока и три блока для проведения вычислений с плавающей запятой, также выполняющих команды VIS. Данный процессор производится по 0,18-микронной технологии и функционирует на тактовых частотах от 750 до 900 МГц. Планируется, что его последующие модели будут работать на частоте 1,5 ГГц. Внутренний кэш данных первого уровня имеет объем 64 Кбайт, а кэш команд — 32 Кбайт. Оба обладают 4-канальной множественно-ассоциативной структурой и состоят из 32-байтовых блоков. Емкость внешнего кэша второго уровня с прямым отображением может быть равной 4 или 8 Мбайт. Процессор UltraSPARC III ориентирован на использование в мультипроцессорных системах, состоящих из сотен процессоров.

Семейство microSPARC

Еще одно семейство процессоров, базирующихся на архитектуре SPARC, называется microSPARC. В него входят 32-разрядные процессоры, основанные на версии 8 спецификации архитектуры SPARC и предназначенные для недорогих однопроцессорных систем. Некоторые из них, такие как microSPARCIIep, обладают интерфейсом PCI и контроллером памяти и подходят для использования во встраиваемых системах. Тот факт, что эти микропроцессоры полностью совместимы с процессорами UltraSPARC, очень важен для разработчиков встроенных систем, поскольку позволяет создавать и тестировать предназначенное для них программное обеспечение на мощных рабочих станциях и лишь на последних стадиях разработки переносить его на целевой процессор.

11.6. Семейство процессоров Compaq Alpha

В 1992 году корпорация Digital Equipment анонсировала выпуск архитектуры Alpha — наследника 32-разрядного семейства VAX. В 1998 году Digital Equipment приобрела компанию Compaq, разработчика линейки высокопроизводительных рабочих станций и серверных систем на основе процессоров Alpha, которые она идентифицировала числами 21X64, где X = 0, 1 и 2.

Архитектура Alpha относится к классу RISC. Процессоры Alpha обрабатывают 64-разрядные адреса и данные и содержат по 32 регистра общего назначения и по 32 регистра с плавающей запятой. Для обеспечения суперскалярного выполнения команд во всех процессорах 21X64 имеется несколько конвейерных операционных блоков, а также осуществляется статическое и динамическое предсказание переходов. Предусмотрены раздельные кэши для команд и данных.

Основная задача разработчиков конвейерных процессоров состоит в упрощении логики обработки на каждой ступени конвейера с целью минимизации времени задержки и соответствующего повышения тактовой частоты. Важной характеристикой процессоров Alpha является то, что для достижения этой цели в них применяются только простые форматы команд и несложные режимы адресации. Команды обмена данными между кэш-памятью первого уровня и процессором выполняются лишь для 32- и 64-разрядных данных и минимизируют задержку при их пересылке.

11.6.1. Форматы команд и режимы адресации

В системе Alpha определено четыре типа команд. Все они имеют длину 32 бита.

- *Операционные.* К этому типу относятся команды, выполняющие целочисленные операции, операции с плавающей запятой и операции с отдельными байтами. Они имеют формат с тремя операндами, содержащимися в регистрах процессора или непосредственно в команде.
- *Операции с памятью.* Данная категория охватывает команды загрузки/сохранения, использующие в процессе работы единственный режим адресации — регистровую адресацию со смещением.
- *Команды перехода.* В командах условного перехода задается величина смещения, определяющая направление и величину перехода по целевому адресу относительно счетчика команд. Специальный регистр кодов условий отсутствует; при необходимости коды условий с помощью операционных команд записываются в регистр общего назначения, который затем указывается в команде перехода. В командах безусловного перехода обновленное значение счетчика записывается в регистр, заданный командой. Так происходит в том случае, если осуществляется переход к подпрограмме и сохраненное значение планируется использовать в качестве адреса возврата.
- *Команды PAL* (Privileged Architecture Library). Выполняют функции операционной системы, недоступные в пользовательском режиме. Эти привилегированные команды осуществляют доступ к аппаратным ресурсам, то есть к регистрам состояния процессора, недоступным для стандартного набора команд. Кроме того, подпрограммы PAL содержат команды, отсутствующие в системе команд Alpha. Это команды обработки прерываний и команды для работы с регистрами блока управления памятью.

11.6.2. Процессор Alpha 21064

Первой реализацией архитектуры Alpha был процессор 21064 — микросхема с тактовой частотой 200 МГц и энергопотреблением 300 Вт, содержащая порядка 1,7 млн. транзисторов. Процессор имеет 8-килобайтовые кэши первого уровня для команд и данных. Оба кэша представляют собой кэш-память с прямым отображением и обрабатывают блоки размером 32 байта. Внешний кэш второго уровня может иметь емкость от 128 Кбайт до 8 Мбайт. Блок управления памятью

содержит отдельные буферы быстрого преобразования адресов для команд (12 записей) и данных (32 записи).

За один такт в исполнительные блоки процессора Alpha могут загрузиться максимум две команды. Всего используется четыре независимых операционных блока: целочисленный, с плавающей запятой, блок обработки переходов и блок загрузки/сохранения. Конвейеры этих блоков состоят из 7, 10, 6 и 7 ступеней соответственно. Первые четыре ступени являются общими и пригодны для параллельной обработки двух потоков команд.

11.6.3. Процессор Alpha 21164

Процессор 21164 был выпущен в 1994 году. Он характеризовался вдвое большей производительностью, чем 21064, состоял из 9,3 млн. транзисторов, потреблял 50 Вт и работал на тактовой частоте 300 МГц. Кроме 8-килобайтовых кэшей первого уровня для команд и данных на микросхеме этого процессора располагается кэш второго уровня объемом 96 Кбайт, общий для команд и данных. Кэш второго уровня имеет 3-канальную множественно-ассоциативную структуру и состоит из 64-байтовых блоков. Емкость внешнего кэша третьего уровня может изменяться от 1 до 64 Мбайт.

Максимальная скорость выполнения команд составляет 4 команды за такт, что вдвое выше, чем у процессора 21064. Кроме того, в процессор 21164 добавлено еще одно функциональное устройство, предназначенное для управления кэшами L2 и L3. Количество ступеней конвейеров у сравниваемых процессоров одинаковое. Среди аппаратного обеспечения, управляющего памятью, — буфер быстрого преобразования адресов на 48 записей для доступа к командам и буфер на 64 записи для доступа к данным.

11.6.4. Процессор Alpha 21264

Процессор Alpha 21264 завершает линейку 21X64. Этот процессор с тактовой частотой 500 МГц был выпущен в 1998 году. В начале 2001 года появилась его версия с тактовой частотой 850 МГц. Микросхема процессора 21264 содержит порядка 15 млн. транзисторов.

В плане организации кэш-памяти этот процессор значительно отличается от своих предшественников. Кэши для команд и данных существенно увеличены и имеют объем по 64 Кбайт. Каждый обладает 2-канальной множественно-ассоциативной структурой. Общий кэш второго уровня является внешним и занимает от 1 до 16 Мбайт. Хотя внутренний кэш второго уровня отсутствует, повышение частоты попадания и увеличенный кэш первого уровня снижают общее время задержки при доступе к памяти.

Еще одно важное отличие процессора 21264 от предшественников заключается в его способности направлять команды в операционные блоки не в том порядке, в котором они упоминаются в программе (см. главу 8), благодаря чему увеличивается количество параллельно выполняемых команд типичных программ. При этом максимальное число команд, реализация которых может завершаться за один такт, остается таким же, как у процессора 21164, — равно четырем.

Операционные блоки тоже претерпели изменения. Заменены некоторые элементы целочисленного блока, а также блока с плавающей запятой, добавлен новый блок для обработки видеоданных. Такое увеличение объема аппаратного обеспечения и тактовой частоты наряду с внеочередным выполнением команд привели к тому, что производительность процессора 21264 по сравнению с процессором 21164 увеличилась вдвое.

11.7. Семейство процессоров Intel IA-64

В середине 1990-х годов Intel и Hewlett-Packard приступили к совместной разработке микропроцессорной архитектуры под названием IA-64. Первый процессор, в котором она была реализована, назывался Itanium (первоначально он имел кодовое название Merced). Архитектура IA-64 кардинально отличается от IA-32, лежащей в основе 32-разрядных процессоров Intel с версии 80386 по Pentium. Корпорация Intel намеревается продолжить выпуск процессоров на основе обеих архитектур — как IA-32, так и IA-64.

В архитектуре IA-64 определены 64-разрядное адресное пространство и 64-разрядные форматы целых чисел, а также чисел с плавающей запятой. Для записи 3-регистровых команд типа RISC требуется 41 бит. Они состоят из 7-битовых регистровых полей, необходимых для доступа к 128 регистрам общего назначения и 128 регистрам с плавающей запятой, 6-битового поля, в котором задается условие выполнения команды, и 14 битов, где определяется код операции.

11.7.1. Блоки команд

Отличительной особенностью архитектуры IA-64 является то, что 41-битовые команды объединяются в 128-битовые *блоки* (bundle) по три команды в каждом. Любой такой блок имеет дополнительное 5-битовое поле, называемое *шаблоном*, в которое компилятор записывает информацию о параллельном выполнении команд. Например, один из кодов шаблонов соответствует отметке *stop*, служащей признаком конца группы команд, которые могут выполняться параллельно. Такая группа может состоять из нескольких блоков команд. Информация шаблона нужна процессору для планирования параллельной реализации команд в нескольких функциональных блоках, то есть для организации суперскалярного выполнения команд. Описанная функция архитектуры IA-64 называется EPIC (Explicitly Parallel Instruction Computing — явное параллельное выполнение команд). Небезосновательным будет утверждение, что EPIC представляет собой расширение концепции построения систем команд, получившей называние VLIW (Very Long Instruction Word — очень длинное слово команды). В архитектурах VLIW каждая команда определяет несколько разных операций с независимыми операндами-данными, которые могут выполняться параллельно.

11.7.2. Условное выполнение

Интересной архитектуру IA-64 делает технология условного выполнения команд, называемая *предикацией*. В каждой команде 6-разрядное поле *предиката* задает один из 64 однобитовых *флагов-предикатов* процессора. Эти флаги играют

ту же роль, что и флаги кодов условий в традиционных процессорах. Если указанный в команде флаг равен 1, команда выполняется, иначе — нет. На самом деле команда проходит через конвейер команд, но ее результаты записываются по адресу назначения только в том случае, если флаг предиката равен 1. Описанная функция подобна условному выполнению команд в архитектуре ARM, о которой говорилось в первой части главы 3.

В ряде случаев благодаря условному выполнению команд ускоряется реализация самой программы, за счет отсутствия условных переходов. Например, вместо команды условного перехода на несколько команд вперед можно условно выполнить блок команд до точки перехода. Флаг перехода, управляющий выполнением каждой команды блока, устанавливается командой проверки или сравнения, расположенной в начале блока.

Описанный выше способ позволяет, в частности, увеличить быстродействие при генерировании кода IA-64 для конструкции if-then-else. На рис. 11.3, *а* показан стандартный машинный код, выполняющий команду сложения, если значения регистров R1 и R2 равны, и команду вычитания в противном случае. На рис. 11.3, *б* представлен соответствующий код IA-64. Команда IA-64 Compare-Equal работает следующим образом. Если содержимое регистров R1 и R2 одинаково, флаг предиката P1 устанавливается в 1, иначе — в 0. Значение флага P2 делается равным дополнению до P1. Если P1 = 1, активизируется команда Add, а если P2 = 1, то вызывается команда Subtract. Двойные точки с запятой служат для указания позиций останова. В этом примере команды Add и Subtract между двумя парами точек с запятой могут выполняться параллельно. В регистр назначения будет записан результат только одной из них, какой именно, зависит от значений флагов предикатов P1 и P2. Если значения P1 и P2 будут определены до того, как команды Add и Subtract достигнут ступеней записи, конвейеры будут работать без остановки. Кроме описанных функций, обеспечивающих повышение производительности, в процессорах IA-64 применяются технологии предсказания переходов и упреждающего выполнения команд (см. главу 8).

```
        Compare          R1,R2
        Branch≠0         ELSE
THEN:   Add              R3,R4,R5
        Branch           NEXT
ELSE:   Subtract         R6,R7,R8
NEXT:   ...
```

а

```
        CompareEqual     P2,P1 = R1,R2 ;;
        (P1) Add         R3,R4,R5
        (P2) Subtract    R6,R7,R8 ;;
NEXT:   ...
```

б

Рис. 11.3. Реализация конструкции if-then-else в архитектуре IA-64: обычный код (*а*); код IA-64 (*б*)

11.7.3. Упреждающая загрузка

С целью уменьшения задержек на выполнение команд загрузки данных в регистры, которым требуется обращение к основной памяти, компилятор может выбрать специальную форму загрузки — *упреждающую загрузку*. Команды упреждающей загрузки размещаются в программе до того места, в котором при обычном ходе ее выполнения должна была быть произведена загрузка данных. В результате повышается вероятность того, что к моменту, когда потребуются данные из регистра, они уже будут загружены в него. Перед использованием этих данных следует произвести проверку того, действительно ли они загружены в регистр. Особая аккуратность требуется при помещении команды упреждающей загрузки в программу, если за ней будет следовать команда прогнозированного перехода.

11.7.4. Регистры и стек регистров

В архитектуре IA-64 определено 128 регистров общего назначения, пригодных для хранения чисел с плавающей запятой двойной точности (64 бита). Два числа одинарной точности могут быть упакованы в один регистр. Кроме того, имеется восемь 64-битовых регистра для хранения адресов вызова/возврата из подпрограмм.

Первые 32 из 128 регистров общего назначения, от R0 до R31, применяются как обычные регистры для хранения адресов и данных. Остальные 96 регистров, от R32 до R127, составляют *стек регистров*, предназначенный для хранения локальных переменных подпрограмм и параметров, пересылаемых между вызываемыми и вызывающими подпрограммами. Этот стек регистров заменяет стек процессора, обычно реализуемый в памяти, о чем рассказывалось в разделе 2.9.1. Управление стеком регистров осуществляется таким образом, что при вызове вложенных подпрограмм не приходится сохранять регистры в памяти и восстанавливать их из нее. Это означает, что общее количество регистров, предназначенных для хранения локальных переменных и параметров всех подпрограмм, не должно превышать 96. В противном случае управляющее аппаратное обеспечение процессора автоматически «сбросит» часть стека регистров в память для освобождения необходимого регистрового пространства, а по возвращении из подпрограммы восстановит ее из памяти.

Переименованием регистров также автоматически управляет процессор, так что все процедуры (основная программа и вызванные ею подпрограммы) всегда содержат ссылки только на свои локальные регистры, начиная с R32, даже если реальные физические регистры иные. (В главе 8 обсуждался еще один способ переименования регистров.) Для передачи параметров применяется область общих регистров вызывающей и вызываемой подпрограмм.

На рис. 11.4 приведен пример управления стеком регистров при вызове подпрограммы из главной программы. На рисунке регистры представлены в виде стека, направленного вверх. Вам легко будет сравнить его со стеками из главы 2, в которых данные размещены в порядке уменьшения адресов памяти. Регистры от R0 до R31 доступны всем подпрограммам и могут рассматриваться как область для хранения глобальных переменных. На рисунке они не изображены. Предполагается, что главная программа использует для хранения своих локальных переменных

восемь регистров, от R32 до R39, а еще четыре регистра, от R40 до R43, применяются для передачи параметров подпрограмме. Перечисленные регистры объявляются в главной программе при помощи следующей команды:

```
Alloc 8,4
```

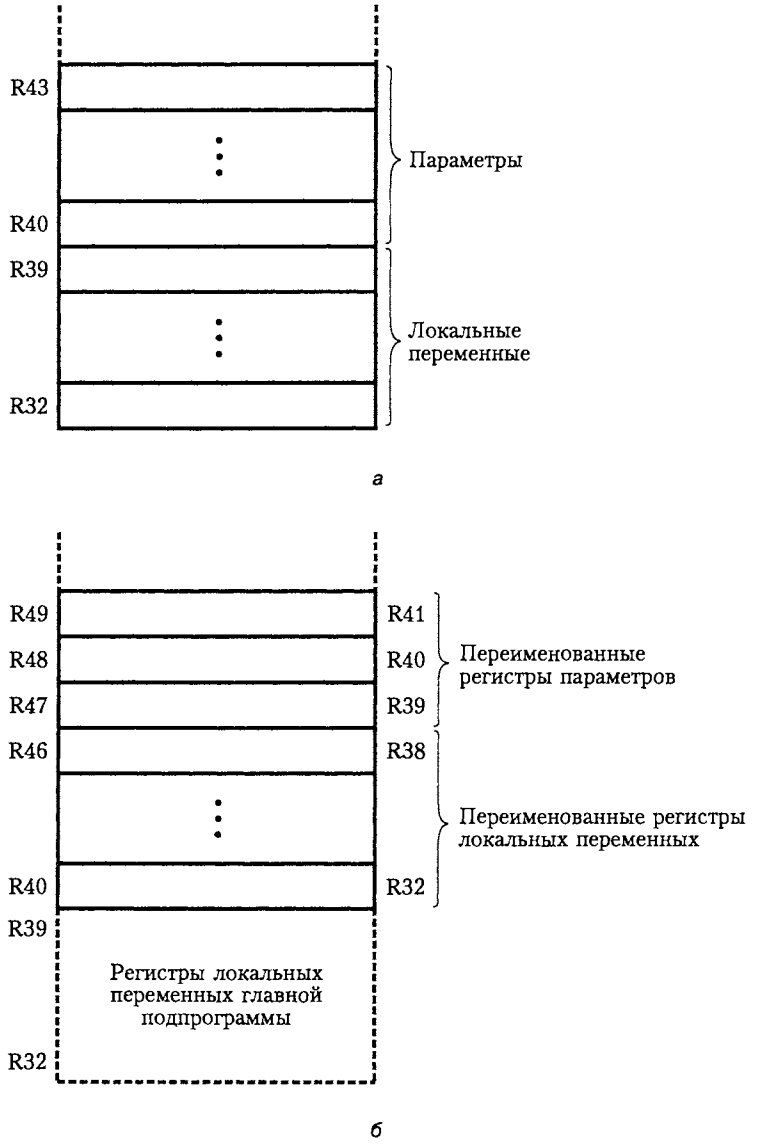


Рис. 11.4. Выделение пространства в стеке регистров для локальных переменных и передачи параметров в архитектуре IA-64: активная область стека регистров в главной программе после выполнения команды Alloc 8,4 (*а*); активная область стека регистров в подпрограмме после выполнения команды Alloc 7,3 (*б*)

Данное объявление соответствует ситуации, смоделированной на рис. 11.4, *а*. Получив вызов от главной программы, подпрограмма выполнит команду

Alloc 7,3

и объявит, что ей нужно восемь локальных регистров, включая четыре регистра, используемых для получения параметров и передаваемых вызывающей процедурой, а также три регистра для передачи параметров второй подпрограмме. Десять физических регистров, от R40 до R49, переименовываются аппаратным обеспечением процессора, чтобы подпрограмма могла ссылаться на них как на регистры от R32 до R41. Часть стека регистров (рис. 11.4, *б*) активна во время выполнения подпрограммы. Информацией, необходимой для автоматического переименования регистров, располагает команда Alloc, активизируемая главной программой. Когда подпрограмма инициирует возврат в главную программу, активная часть стека регистров возвращается в состояние, показанное на рис. 11.4, *а*. В описанном стеке регистров реализована одна из версий концепции *регистровых окон*, используемой в RISC-архитектуре Беркли и в процессоре UltraSPARC II (см. главу 8).

В архитектуре IA-64 используется еще одна форма переименования регистров, называемая *ротацией регистров*. Она предназначена для выполнения последовательных итераций цикла в тех случаях, когда отсутствует зависимость между данными, обрабатываемыми на последовательных итерациях. Обычно в последовательных итерациях цикла применяются одни и те же регистры, указанные в теле цикла. В архитектуре IA-64 компилятор генерирует для одной копии тела цикла такой код, который позволяет аппаратному обеспечению автоматически переименовывать регистры, чтобы на разных итерациях цикла использовались различные аппаратные регистры. Это позволяет начинать очередную итерацию цикла до завершения предыдущей. Описанная технология повышения скорости выполнения циклов называется программной конвейеризацией. Она отличается от технологии развертывания циклов, при которой в машинный код программы помещаются копии тела цикла.

11.7.5. Процессор Itanium

Процессор Itanium является первой реализацией архитектуры IA-64. Он содержит большое количество дублирующихся функциональных блоков, которые рассчитаны на операции различных типов: целочисленные, с плавающей запятой и мультимедийные (подобные операциям MMX в архитектуре IA-32, описанной в главе 3). Суперскалярное выполнение команд в нем настолько эффективно, что на каждом такте при частоте 800 МГц в 10-ступенчатый конвейер может загружаться до шести команд (два 3-командных блока). Набор функциональных блоков этого процессора таков: 4 целочисленных блока, 4 блока с плавающей запятой, 4 мультимедийных блока (MMX), 2 блока загрузки/сохранения и 3 блока обработки переходов. Банк целочисленных регистров имеет 8 портов чтения и 6 портов записи, обеспечивающих одновременный обмен данными с несколькими функциональными блоками.

Процессор имеет кэш-память трех уровней. Кэши первого и второго уровней располагаются на той же микросхеме, что и процессор, а кэш третьего уровня — на

отдельной микросхеме, упакованной в один картридж с процессором. Кэш-память первого уровня включает раздельные кэши команд и данных объемом по 16 Кбайт каждый. Оба имеют 4-канальную множественно-ассоциативную структуру и состоят из 32-байтовых блоков. Кэш команд может предоставить процессору один или два 3-командных блока за один такт (256 бит). Объем кэша L2 составляет 96 Кбайт. Он обладает 6-канальной множественно-ассоциативной структурой и состоит из 64-байтовых блоков. Кэш L3 имеет объем 4 Мбайт и 4-канальную множественно-ассоциативную структуру, состоит из 64-байтовых блоков и взаимодействует с кэшем L2 через 128-разрядное внутреннее соединение, работающее на тактовой частоте процессора и обеспечивающее пересылку данных со скоростью 12,8 Гбайт/с.

Взаимодействие между кэшами и процессором организовано таким образом, чтобы предельно уменьшить негативный эффект остановов конвейера и промахов при обращении к кэш-памяти. Некоторые аспекты этого взаимодействия заслуживают отдельного анализа. Между кэшем команд первого уровня и процессором есть специальный буфер, вмещающий до восьми 3-командных блоков. Он позволяет продолжать выборку команд из кэша L1 даже тогда, когда конвейер процессора останавливает их загрузку. Более того, процессор может считывать команды из буфера и в случае промаха во время опережающей выборки. Аналогичный буфер, поддерживающий опережающую выборку команд из L2 в L1, имеется между кэшами L1 и L2. По размеру он вдвое превышает буфер между процессором и кэшем L1. Кэш L1 содержит данные только для банка целочисленных регистров. Операнды с плавающей запятой загружаются в соответствующие регистры непосредственно из кэша L2.

Аппаратный блок, включающий процессоры и кэши, соединяется с остальными компонентами системы, такими как основная память и устройства ввода-вывода, посредством 64-разрядной системной шины. Шина способна работать на тактовой частоте 266 МГц и обеспечивает скорость пересылки данных 2,1 Гбайт/с.

Контроллер внешней шины поддерживает мультипроцессорную конфигурацию с четырьмя процессорами Itanium. Этот контроллер управляет операциями согласования кэшей, необходимыми в тех случаях, когда в кэш-памяти нескольких процессоров содержатся общие данные (см. главу 12).

11.8. Стековый процессор

В рассмотренных процессорах для хранения операндов применяются регистры общего назначения. Несколько лет назад компания Hewlett-Packard выпустила компьютер HP3000, главной архитектурной особенностью которого была система команд, ориентированная на обработку операндов, хранящихся в стеке. Доступ к операндам в стеке ограничивался только верхним элементом стека, а результаты всегда помещались только в вершину стека. Такой способ организации данных не свойственен современным процессорам с архитектурой RISC или CISC, которые характеризуются значительной степенью параллелизма. В этих процессорах для обеспечения более высокой производительности необходим одновременный

доступ к нескольким операндам, хранящимся в большом наборе регистров. Тем не менее процессор HP3000, а также ранние модели компьютеров B5500, B6500 и B6700 производства Burroughs Corporation, также ориентированные на обработку данных в стековых структурах, важны с исторической точки зрения как коммерческие реализации стекового принципа вычислений. В этом разделе акцент сделан только на тех характеристиках указанных компьютеров, которые связаны с организацией стека.

11.8.1. Структура стека

Компьютер HP3000 имеет 16-разрядную организацию. В его памяти команды программ и данные упорядочены в виде отдельных доменов; они не могут смешиваться, за исключением тех случаев, когда в программах присутствуют непосредственно заданные операнды-данные. В качестве указателей на сегменты программы и данных применяются аппаратные регистры (рис. 11.5).

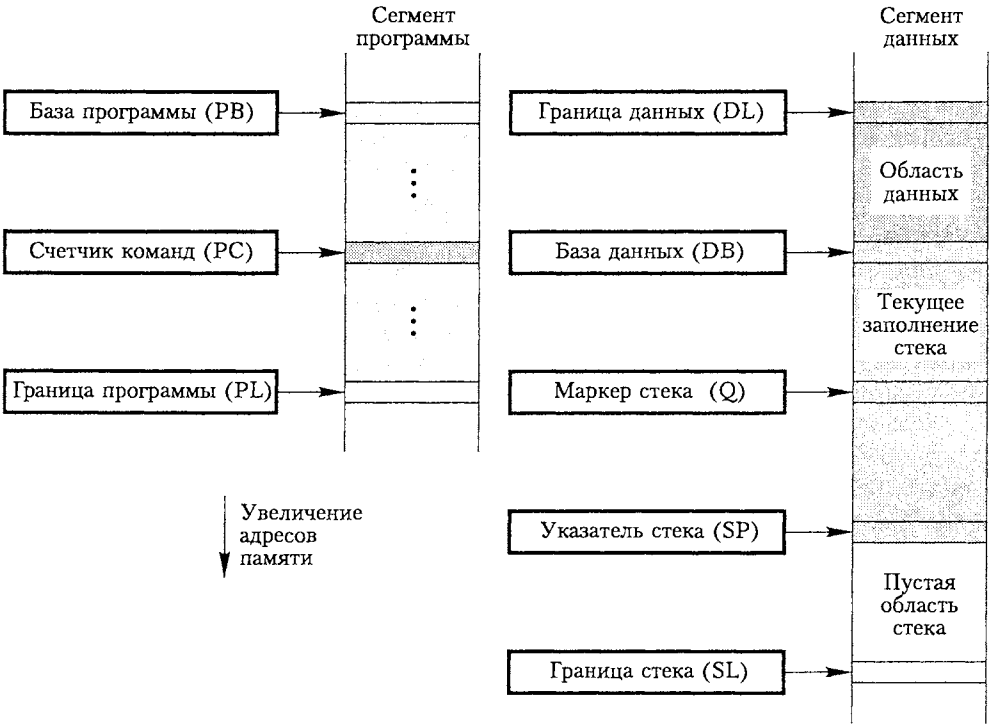


Рис. 11.5. Организация сегментов программы и данных в процессоре HP3000

Сегмент программы определяется тремя регистрами. Регистры базы программы (PB) и границы программы (PL) задают занимаемую программой область памяти, а счетчик команд (PC) указывает на текущую команду. Каждый регистр содержит соответствующий 16-разрядный адрес.

Сегмент данных делится на две части — стек и область данных. Для доступа к этому сегменту служат пять 16-разрядных указателей. Содержимое регистра базы данных (DB) указывает на начало стека. Стек растет в направлении увеличения адресов. Если верхний элемент стека располагается по адресу i, то следующий элемент будет помещен в стек по адресу $i + 1$. Напомним, что все стеки, обсуждавшиеся ранее в этой книге, увеличивались в противоположном порядке, то есть в направлении уменьшения адресов. Адрес верхнего элемента стека, называемый также вершиной стека, хранится в 16-разрядном регистре указателя стека (SP). Хотя, как рассказывается ниже, SP на самом деле не является аппаратным регистром, его можно представлять как регистр. Когда элементы данных помещаются в стек или извлекаются из него, указатель стека увеличивается или уменьшается. С точки зрения пользователя он функционирует как любой другой 16-разрядный регистр-указатель. Верхняя граница стека определяется содержимым регистра границы стека (SL). Таким образом, стек может расти до тех пор, пока содержимое [SP] не станет равным содержимому [SL]. Любые попытки превысить объем стека, ограниченный значениями DB и SL, предотвращаются аппаратным обеспечением. Область данных располагается от адреса, заданного в регистре DB, до адреса, обозначенного в регистре границы данных DL.

Итак, регистры-указатели определяют текущий размер стека, его максимальный размер и расположение в памяти. Стек является динамической структурой, которая легко изменяется. На рис. 11.5 показан еще один указатель, регистр маркера стека Q. Этот регистр задает начальную точку стека данных текущей процедуры. Другими словами, регистр Q указывает на четвертое слово записи стека (*маркера стека*), состоящей из четырех слов и используемой для передачи управления от процедуры к процедуре. Регистр Q играет практически ту же роль, что и регистр указателя стекового фрейма, о котором рассказывалось в главе 2. Если выполнение процедуры должно быть приостановлено, предположим из-за прерывания, информация, необходимая для правильного возврата к этой процедуре, помещается в стек в форме маркера стека.

Первое слово маркера стека хранит текущее содержимое индексного регистра, а второе — адрес возврата. Адрес возврата определяется как разность между значением регистра PC, указывающего на следующую команду текущей подпрограммы, которая должна быть выполнена процессором, и значением регистра PB. Благодаря тому что в маркере стека хранится относительное, а не абсолютное значение адреса возврата, программы можно перемещать в памяти, изменяя значение в регистре PB. В третьем слове маркера стека находится информация, содержащаяся в регистре состояния, а в четвертом слове — значение расстояния между текущим и предыдущим маркерами стека.

На рис. 11.6 маркер стека k помещен в стек при вызове процедуры Procedure_k, а еще один маркер стека, $k + 1$, добавлен в стек во время вызова следующей процедуры, Procedure_{k+1}. Когда выполнение новой процедуры завершается, машина передает управление предыдущей процедуре, используя данные маркера стека $k + 1$. В то же время регистр Q должен быть установлен таким образом, чтобы указывать на четвертое слово маркера стека k. Это легко реализовать, поскольку информация об интервале между маркерами стека хранится в каждом маркере. Кроме

того, в регистр SP заносится указатель на элемент, непосредственно предшествующий маркеру стека $k + 1$. В результате SP указывает на вершину стека, используемого процедурой Procedure_k, и восстанавливается состояние конфигурации на момент вызова процедуры Procedure_{k+1}. Эта технология может применяться для обработки вложенных вызовов любого количества процедур. Обмен параметрами между процедурами также осуществляется через стек.

Наряду с регистрами-указателями в компьютерах HP3000 имеются и другие аппаратные регистры. Программисту видимы только два из них — индексный регистр и регистр состояния. Оба функционируют так же, как и соответствующие регистры в большинстве других машин. Обратите внимание на то, что программисту недоступны регистры общего назначения. Вместо них для операций с данными применяется стек, о чем вы узнаете из следующего раздела.

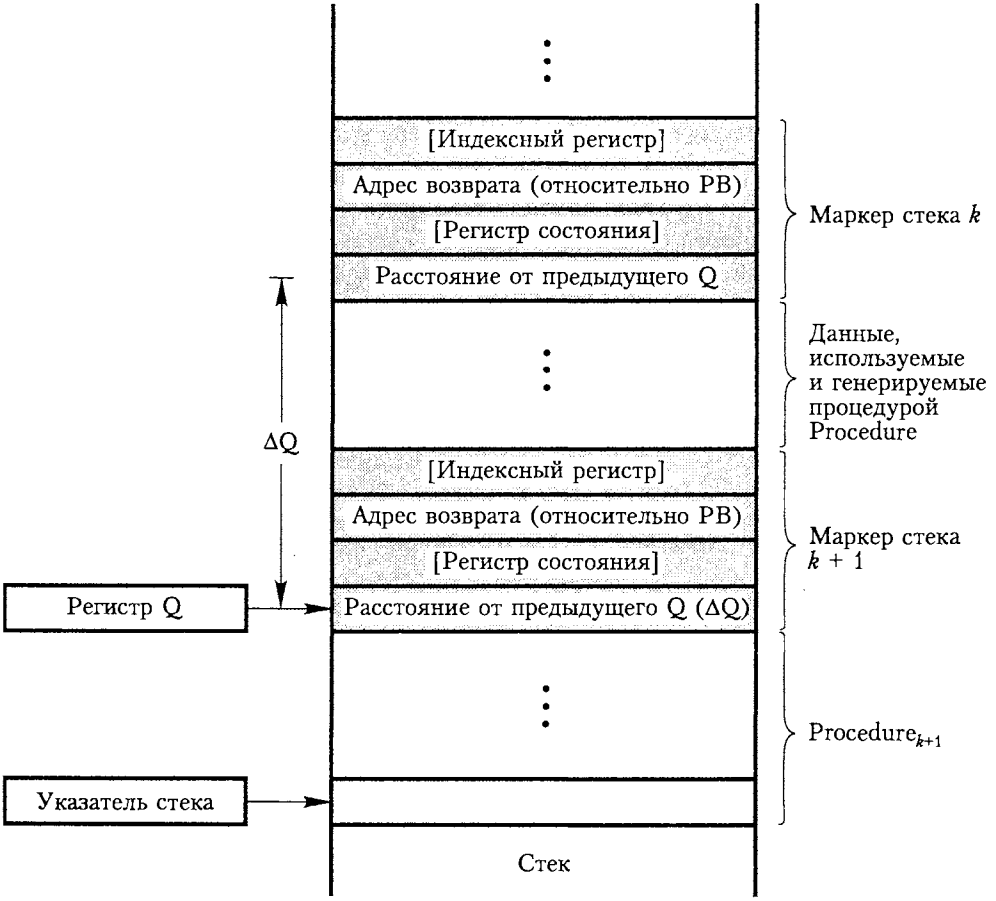


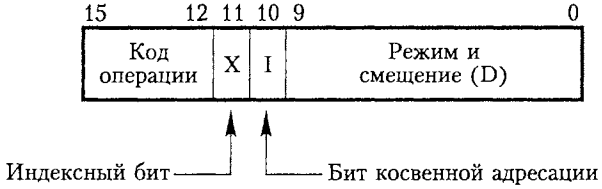
Рис. 11.6. Маркеры стека в компьютере HP3000

11.8.2. Стековые команды

Базовая стратегия стековых компьютеров заключается в выполнении операций над данными, занимающими несколько верхних элементов стека. Генерируемые этими операциями результаты также сохраняются в стеке. Существуют команды, обеспечивающие пересылку данных между стеком и памятью.

В компьютере HP3000 определено множество команд, и все они имеют длину 16 бит. Большинство команд так или иначе обращаются к стеку. Как правило, в нем располагаются операнды, их адреса и другие смежные данные. Это обеспечивает высокую гибкость при использовании 16-разрядных кодов команд. Различают 13 основных классов команд. Мы не станем описывать полную систему команд HP3000, а ограничимся рассмотрением тех классов, которые связаны со стековой организацией компьютера. Для начала изучим команды класса Memory Address, формат которых представлен на рис. 11.7. Здесь насчитывается 11 различных значений 5-битового поля кода операции. К классу Memory Address принадлежат следующие команды:

- LOAD — помещает в стек заданное слово памяти;
- STOR — выталкивает верхнее слово стека в указанное место памяти;
- ADDM — прибавляет указанное слово памяти к значению вершины стека и заменяет это значение полученной суммой;
- MPYM — умножает указанное слово памяти на значение вершины стека и заменяет это значение младшим словом произведения;
- INCM — выполняет операцию инкремента заданного слова памяти.



Режим		Битовый код $b_9\ b_8\ b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$	Исполнительный адрес памяти
PC +	относительный	0 0 ← D →	[PC] + D
PC −	относительный	0 1 ← D →	[PC] − D
DB +	относительный	1 0 ← D →	[DB] + D
Q +	относительный	1 1 0 ← D →	[Q] + D
Q −	относительный	1 1 1 0 ← D →	[Q] − D
SP −	относительный	1 1 1 1 ← D →	[SP] − D

Рис. 11.7. Формат команд класса Memory Address процессора HP3000

Операнд этих команд хранится в памяти и задается в относительном режиме: его адрес указывается относительно содержимого регистра PC, DB, Q или SP. Как показано на рис. 11.7, 10-разрядное поле режима и смещения содержит код адресации и значение смещения. Смещение в разных режимах неодинаково, поэтому и поле смещения имеет длину от 6 до 8 разрядов. Биты индексного и косвенного режимов адресации определяют, применяется в данной команде адресация определенного типа, индексная либо косвенная, или же используются оба режима адресации. Мы указали лишь те режимы, которые могут использоваться для адресации операндов в области данных (рис. 11.5).

Ко второму классу принадлежат команды пересылки, содержащие ссылки на один или два операнда в памяти. Эти команды пересылают слова или байты из одной области памяти в другую, сравнивают две строки байтов в памяти или сканируют байтовые строки в поиске заданного значения байта. Адреса памяти вычисляются в относительном режиме. Смещение не задается явно в команде, в стек помещаются соответствующие данные. Следует учитывать, что адреса можно задавать только относительно базы программы или данных, то есть относительно содержимого регистра PB или DB. Примером команды этого класса служит базовая команда MOVE, пересылающая k слов из одного места памяти в другое. Изучим досконально эту команду.

- Значение k задается в верхнем элементе стека.
- Во втором элементе стека указывается адрес первого исходного слова в памяти относительно содержимого регистра PB или DB.
- В третьем элементе стека задается адрес места назначения в памяти относительно содержимого регистра PB или DB.

Команда MOVE может быть представлена 16-битовым кодом, поскольку информация об адресах и длине данных содержится в стеке, а не в самой команде. Эта информация должна быть помещена в стек перед выполнением команды.

Теперь рассмотрим команды класса Stack, формат которых вы видите на рис. 11.8. Этот класс команд идентифицируется четырьмя нулями в старших разрядах. Остальные 12 разрядов используются для задания конкретной команды. Они разделены на два 6-разрядных поля, каждое из которых может применяться для отдельной операции. Оперируя 6 битами, можно задать 64 различные стековые операции. В команде, определяющей одну стековую операцию, насчитывается 10 разрядов (главный код операции плюс код стековой операции A); оставшиеся 6 разрядов игнорируются. Если в остальных разрядах задана вторая стековая операция (код стековой операции B), то она осуществляется по завершении первой. В одну команду могут быть упакованы две стековые операции. Такое эффективное использование пространства кода команды возможно только потому, что адресные данные и операнды не включаются непосредственно в команду.

Приведем несколько примеров команд класса Stack:

- ADD — складывает содержимое верхних двух слов стека, удаляет их из стека и проталкивает сумму в стек;
- CMP — сравнивает содержимое верхних двух слов стека, устанавливает соответствующим образом флаги кодов условий и удаляет оба слова из стека;

- DIV — делит целое число, хранящееся во втором слове стека, на целое число, которое находится на вершине стека, после чего второе слово стека заменяет частным, а первое — остатком;
- DEL — удаляет верхнее слово стека.

Рассматриваемый класс включает множество других команд, в частности те, что выполняют гораздо более сложные операции. Так, команда Divide Long (DIVL) делит целое число, которое представлено двойным словом, состоящим из второго и третьего элементов стека, на целое число, представленное первым элементом стека. Затем перечисленные три слова удаляются из стека, а остаток и частное помещаются в стек и становятся его первым и вторым элементами. При обсуждении стековых команд термин *команда* является не совсем точным. Правильно было бы говорить о конкретных операциях, например об операциях сложения и деления, поскольку в одной команде могут быть заданы две операции. Однако выбранный нами путь привычнее. Упаковка двух команд в одну возможна только в том случае, если обе осуществляют операции со стеком. В остальных случаях код операции B не применяется.

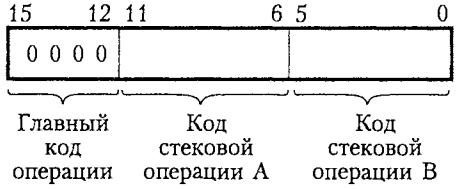


Рис. 11.8. Формат стековых команд в процессоре HP3000

До сих пор обсуждалось только одно достоинство операции упаковки команд, заключающееся в уменьшении объема кода программы. Но есть еще один положительный момент, состоящий в сокращении количества обращений к памяти, поскольку две команды, упакованные в одно 16-разрядное слово, извлекаются из памяти за один раз. Следует помнить, что в ходе выполнения стековых команд осуществляется доступ к операндам, хранящимся в стеке, а для этого приходится обращаться к памяти, так как именно там находится стек.

Чтобы продемонстрировать роль стека как временного хранилища промежуточных результатов при выполнении арифметических операций, рассмотрим простой пример. На рис. 11.9, *a* показано, как вычисляется арифметическое выражение

$$w = \frac{(a+b)}{c/d + (e \times f)/(g+h)}$$

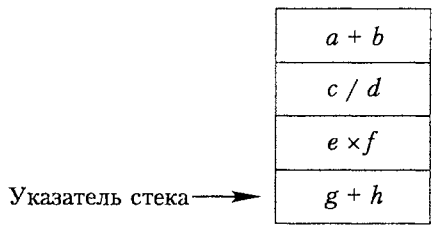
Предположим, что значения переменных *a*, *b*, ... *h* хранятся не на вершине стека, а в памяти по адресам A, B, ... H, и для доступа к ним может использоваться механизм адресации, показанный на рис. 11.7. Все операнды являются целыми числами, а размеры их таковы, что результаты умножения имеют одинарную точность. На рисунке показаны 13 этапов, из которых состоит процесс вычисления приведенной формулы. Операции записаны в порядке сканирования числителя и знаменателя дроби слева направо. Верхний элемент стека обозначен как S.

Таким образом, операция S ← [S] + [B] означает, что содержимое верхнего элемента стека складывается с операндом B, а сумма заменяет значение верхнего элемента стека. Операцию S ← [S − 1] / [S] можно трактовать следующим образом: содержимое второго элемента стека делится на содержимое вершины стека, два операнда удаляются из стека, а в него помещаются частное и остаток.

Осуществлять нужные вычисления позволяют машинные команды, перечисленные на рис. 11.7 (все они уже были описаны в этом разделе). Большинство шагов, за исключением операции деления, могут быть выполнены с помощью одной команды. Команда DIV заменяет делимое и делитель частным и остатком. Поскольку нас интересует только частное, мы удаляем остаток из стека посредством команды DEL. Как рассказывалось выше, любые две идущие подряд стековые команды могут быть объединены в одну 16-разрядную команду. Промежуточные результаты команд сохраняются в стеке. На рис. 11.9, *б* вы видите верхние элементы стека после выполнения шага 9.

Шаг	Выполняемая операция	Машинная команда		
1	S ← [A]	LOAD	A	
2	S ← [S] + [B]	ADDM	B	
3	S ← [C]	LOAD	C	
4	S ← [D]	LOAD	D	
5	S ← [S − 1] / [S]	DIV		} Объединяются
		DEL		
6	S ← [E]	LOAD	E	
7	S ← [S] × [F]	MPYM	F	
8	S ← [G]	LOAD	G	
9	S ← [S] + [H]	ADDM	H	
10	S ← [S − 1] / [S]	DIV		} Объединяются
		DEL		
11	S ← [S − 1] + [S]	ADD		
12	S ← [S − 1] / [S]	DIV		} Объединяются
		DEL		
13	W ← [S]	STOR	W	

а



б

Рис. 11.9. Использование стека при вычислении выражения $w = (a + b)/ [c/d + (e \times f)/(g + h)]$: выполняемые операции и соответствующие машинные команды (*а*); временные результаты, сохраненные в стеке после шага 9 (*б*)

11.8.3. Аппаратные регистры в стеке

Больше всего на скорости выполнения программ сказываются операции доступа к памяти. Чтение операнда команды из памяти осуществляется гораздо дольше, чем чтение из регистра процессора. Поэтому в процессор включают регистр общего назначения и кэш-память. В случае стековых компьютеров для временного хранения данных вместо регистров общего назначения используется стек, который целиком располагается в памяти.

Реализация всего стека при помощи аппаратных регистров — решение, во-первых, дорогостоящее, а во-вторых, недостаточно гибкое. Однако возможен компромисс, при котором большая часть стека располагается в памяти, а несколько его верхних элементов — в регистрах процессора. При такой архитектуре время доступа к стеку значительно сокращается, поскольку наиболее часты обращения именно к верхним элементам стека. В компьютере HP3000 четыре верхних элемента стека содержатся в четырех регистрах процессора.

Если в стек включены аппаратные регистры, то, скорее всего, настоящей вершиной стека является один из регистров. Это означает, что SP не обязательно указывает на память. Чтобы обеспечить возможность отслеживать местонахождение верхних элементов стека в каждый момент времени, указатель SP реализован в виде двух регистров. В 16-разрядном регистре стека, находящегося в памяти (SM), содержится адрес верхнего элемента стека, а 3-битовый регистр SR показывает, сколько верхних элементов стека в настоящий момент насчитывается в аппаратных регистрах — нуль, один, два, три или четыре. Таким образом, значение [SP] представляет собой сумму значений двух регистров:

$$[SP] = [SM] + [SR]$$

Это значение равно адресу в памяти, по которому должен располагаться верхний элемент стека при условии, что все элементы находятся в памяти (рис. 11.10). Программист не должен знать о том, что часть элементов стека присутствует в аппаратных регистрах. Он считает, что весь стек располагается в памяти и существует единственный указатель стека — SP.

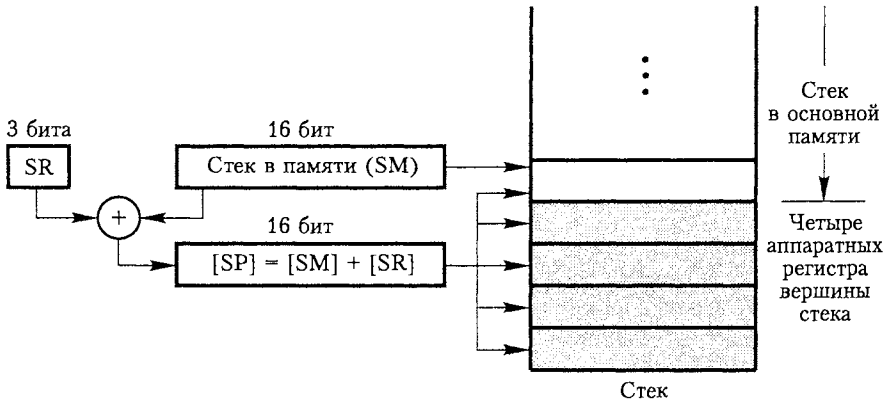


Рис. 11.10. Структура верхней части стека процессора HP3000

11.9. Резюме

Все высокопроизводительные процессоры содержат внутренние кэши для команд, данных и операций преобразования адресов. Они включают несколько независимых конвейерных исполнительных блоков, в которые за один такт может поступать более одной команды. Еще больше повысить производительность позволяет использование сложных методов динамического предсказания переходов и упреждающего выполнения альтернативных ветвей программы, а также применение оптимизирующих компиляторов, транслирующих программы на языках высокого уровня в эффективный код на машинном языке.

Упражнения

11.1. Как условное выполнение команд в архитектуре ARM (см. главу 3) соотносится с предикативным выполнением команд в архитектуре IA-64?

11.2. 16-разрядное подмножество Thumb набора команд ARM предназначено для компактного кодирования программ. Оцените количество команд Thumb, необходимое для написания программы, вычисляющей значение арифметического выражения (рис. 11.9). Предполагается, что в подмножестве Thumb имеется соответствующая команда деления (на самом деле ее нет). Как это оценочное количество команд соотносится с количеством команд программы HP3000, показанной на рис. 11.9, *а*?

11.3. Каковы сходства и различия между семействами процессоров Motorola 680X0 и Intel 80X86, включая версии 68040 и 80486?

11.4. В микропроцессоре 68030 имеется 256-килобайтовый кэш команд и 256-килобайтовый кэш данных. Какой вариант лучше: этот или же предполагающий наличие только 512-килобайтового кэша команд (без кэша данных)? Каковы достоинства и недостатки обоих вариантов? Ответьте на эти вопросы, если в наличии имеется 512-килобайтовый общий кэш для команд и данных.

11.5. Процессор Intel IA-32 (см. главу 3) обладает специальными командами для выполнения операций изолированного ввода-вывода. В процессорах Motorola 680X0 используется только ввод-вывод с отображением в память. Каковы преимущества и недостатки обоих методов ввода-вывода?

11.6. Выполните сравнительный анализ и укажите достоинства и недостатки режимов адресации процессоров Motorola 680X0 и Intel 80X86. Как режимы адресации обоих процессоров реализуют перенос программ из одного места памяти в другое, доступ к спискам операндов и манипулирование символьными строками?

11.7. В разделе 11.3.1 рассказывалось о том, что процессор IA-32 может обращаться к памяти четырьмя разными способами. Приведите примеры ситуаций, в которых предпочтителен тот или иной способ организации памяти.

11.8. Напишите программу для процессора ARM, 68000 или IA-32, вычисляющую значение арифметического выражения (рис. 11.9). Сравните получившуюся

программу с приведенной на рисунке с точки зрения количества машинных команд. Предполагается, что в системе команд ARM имеется соответствующая команда деления (на самом деле ее нет).

11.9. В процессорах Alpha при выполнении операций загрузки и сохранения выровненных данных кэш-памятью и процессором используются только 32- и 64-разрядные команды. Разработайте комбинационную логическую схему для 32-разрядной шины данных, обеспечивающую загрузку одного из четырех байтов 32-битового значения в младший байт места назначения.

11.10. Сравните принцип функционирования стека регистров в процессорах IA-64 и стека, описанного в главе 2. Какие элементы в архитектуре IA-64 соответствуют регистрам SP (указатель стека) и FP (указатель стекового фрейма), упоминаемым в главе 2?

11.11. Опишите аппаратное обеспечение, необходимое для поддержки работы команды Alloc X,Y, которая управляет стеком регистров в архитектуре IA-64. Предполагается, что имеются регистры и сумматоры. Как они применяются?

11.12. У процессора Alpha 21264 кэш-память организована иначе, чем у процессора 21164. Сравните организацию кэш этих процессоров. При каких обстоятельствах программы быстрее выполняются процессором 21264 (с учетом только эффекта кэширования)? Отвечая на вопрос, учитывайте не только частоту попадания в кэш.

11.13. Как в компьютере HP3000 вычисляется выражение

$$w = a\left[(b \times c) + (d \times e) + \frac{f \times g}{h \times i}\right]$$

11.14. В компьютере HP3000 процедура Procedure_i генерирует восемь слов данных, DI_1, ..., DI_8, сохраняющихся в стеке. После их помещения в стек, но до завершения Procedure_i, вызывается новая процедура, Procedure_j. Она генерирует еще 10 слов данных, DJ_1, ..., DJ_{10}, которые тоже сохраняются в стеке. Затем вызывается еще одна процедура, Procedure_k, добавляющая в стек 3 слова данных. Каким в результате будет содержимое верхних уровней стека?

11.15. Подберите оптимальный способ вычисления выражения

$$w = (a + b)(c + d) + (d \times e)$$

в компьютерах HP3000, ARM, Motorola 68000 и IA-32. Значения переменных w, a, b, c, d и e хранятся в памяти, но не по последовательным адресам. В компьютере HP3000 используется адресация памяти в относительном режиме DB+, в компьютерах 68000 и IA-32 — абсолютная или прямая адресация, а в компьютере ARM — относительная адресация. Все произведения имеют одинарную длину.

11.16. Каково максимальное количество элементов стека, задействованных при выполнении программы на рис. 11.9?

11.17. Выполните еще раз упражнение 11.16 в расчете на программы для процессора HP3000 из упражнений 11.13 и 11.15.

Глава 12

Большие компьютерные системы

- Мультипроцессорные и мультикомпьютерные системы
- Архитектура мультипроцессорных систем
- Коммутационные и локальные сети
- Организация памяти в мультипроцессорных системах
- Согласованность кэш-памяти для общих данных
- Системы с общей памятью и с передачей сообщений
- Производительность мультипроцессорных систем

Приложения, которые планируется использовать для выполнения значительного объема вычислений, следует устанавливать в системах с высоким быстродействием. Подобные системы часто называют суперкомпьютерами. В настоящее время без применения суперкомпьютеров невозможно представить себе решение задач в таких областях, как метеорология и исследование Мирового океана, строительство и материаловедение, генная инженерия, гидро- и газодинамика, моделирование сложных физических систем и автоматизированное проектирование (САПР). Но ни одна из описанных в предыдущих главах вычислительных систем не относится к классу суперкомпьютеров.

Для создания высокопроизводительных процессоров могут быть задействованы самые передовые технологии разработки быстродействующих схем и наиболее эффективные архитектурные решения, в частности параллелизм вычислений, конвейерная обработка, кэш-память большого объема, чередование адресов основной памяти и раздельные шины для команд и данных. Все перечисленные, а также подобные им решения используются при создании процессоров для рабочих станций. Они ориентированы на повышение производительности системы без значительного увеличения ее стоимости, и результаты их применения действительно впечатляют — современные рабочие станции имеют более высокую производительность, чем суперкомпьютеры всего лишь десятилетней давности.

Но даже такая высокая производительность рабочих станций не удовлетворяет нуждам ряда приложений, требующих значительно больших вычислительных мощностей, поэтому спрос на суперкомпьютеры по-прежнему сохраняется. Одним из возможных подходов к их разработке является создание системы с несколькими очень мощными процессорными устройствами. Как правило, с этой целью используются

самые быстрые из существующих схем, широкие шины для доступа к основной памяти очень большого размера и мощные средства ввода-вывода. Такие компьютеры потребляют значительное количество энергии и требуют специальных систем охлаждения. В приложениях, выполняющих немалые объемы вычислений, суперкомпьютеры должны с максимальной эффективностью обрабатывать так называемые *векторные данные*, то есть одномерные массивы чисел (элементов), рассматриваемые как единое целое. Например, может возникнуть необходимость сохранить в основной памяти векторное значение, находящееся в регистрах, или же произвести какие-либо операции над вектором, являющимся результатом поэлементного сложения двух 64-элементных векторов. Для наиболее эффективного выполнения приложений с многочисленными векторными операциями используются компьютеры векторной архитектуры. Суперкомпьютеры такого класса производятся компаниями Cray (Cray-1, Y-MP и SV1), Fujitsu (VPP5000), Hitachi (SR8000) и NEC (SX-5). Основным недостатком подобных систем является их очень высокая стоимость (куда входит не только цена, но и затраты на сопровождение и обслуживание).

Привлекательной альтернативой использованию специализированных суперкомпьютеров является применение большого количества однопроцессорных рабочих станций. Это можно сделать одним из двух способов. Первый из них заключается в создании системы с эффективными средствами взаимодействия между процессорами, общими модулями памяти и устройствами ввода-вывода. Системы такого типа обычно называются *мультипроцессорными*. Второй способ предполагает создание системы, состоящей из большого количества рабочих станций, объединенных в локальную коммуникационную сеть. Подобные системы называются *распределенными компьютерными системами*. Мультипроцессорные и распределенные компьютерные системы имеют много общего. Первые характеризуются высокой производительностью и высокой стоимостью, вторые же более естественно вписываются в современные компьютерные инфраструктуры предприятий и имеют более низкую цену. В этой главе мы рассмотрим особенности систем обоих типов.

Высокая производительность системы с большим количеством процессоров обеспечивается за счет параллельного выполнения огромного числа операций. Однако трудность эффективного использования указанных систем заключается в том, что не каждое приложение удается разбить на отдельные задачи, которые можно распределить между несколькими процессорами для параллельного выполнения. Для выделения подобных задач, планирования и координирования путей их решения в мультипроцессорной системе необходимо наличие сложного программного обеспечения и специализированных аппаратных средств. Все эти вопросы освещаются в настоящей главе.

12.1. Виды параллельной обработки

Существует много способов параллельного выполнения отдельных частей вычислительной задачи. С некоторыми из них вы уже познакомились в предыдущих главах. Так, для осуществления операций ввода-вывода в большинстве компьютеров

предусмотрены аппаратные средства, обеспечивающие прямой доступ к памяти, то есть передачу данных между памятью и устройствами ввода-вывода без участия процессора. В частности, передача данных между основной памятью и магнитным диском в обоих направлениях может выполняться контроллером DMA параллельно с работой центрального процессора.

Передача блока данных с диска в основную память инициируется процессором, направляющим контроллеру DMA соответствующие команды. После этого процессор возвращается к своей работе и продолжает выполнять вычисления, никак не связанные с инициированной им передачей информации. А в это время контроллер DMA самостоятельно производит передачу данных и по ее завершении информирует об этом процессор с помощью сигнала прерывания. Получив уведомление о том, что информация прочитана с диска в основную память, процессор переключается на выполнение той задачи, в которой она используется.

Приведенный пример иллюстрирует два фундаментальных аспекта параллельной обработки. Во-первых, для того чтобы такой способ обработки можно было применить для решения конкретной задачи, последняя должна включать подзадачи, допускающие параллельное выполнение разными аппаратными компонентами. В нашем примере такими подзадачами являются вычисления, производимые процессором, и операция чтения данных с диска в основную память, выполняемая контроллером DMA. А во-вторых, необходимо наличие средств для активизации и координирования параллельной обработки. В данном примере параллельная обработка активизируется процессором, инициирующим операцию прямого доступа к памяти. Координирование действий процессора и контроллера DMA осуществляется с помощью прерывания, сигнал которого генерируется контролером DMA по завершении операции. В ответ на этот сигнал процессор начинает вычисления с использованием полученных данных.

Мы рассмотрели самый простой случай параллелизма, в котором выполняются только две задачи. В общем случае масштабные вычисления можно разделить на множество параллельно выполняемых частей. Для поддержки таких параллельных вычислений может использоваться несколько аппаратных структур.

12.1.1. Классификация систем параллельной обработки

В соответствии с общей классификацией систем параллельной обработки данных, однопроцессорная компьютерная система называется системой с *одиночным потоком команд и одиночным потоком данных* (Single Instruction stream, Single Data stream, SISD). Выполняемая процессором программа составляет поток команд, а последовательность элементов данных, которые она обрабатывает, составляет поток данных. Возможна другая схема функционирования, при которой один поток команд выполняется множеством процессоров. При этом каждый процессор обрабатывает только собственные данные. Системы с такой архитектурой именуются системами *с одиночным потоком команд и множественным потоком данных* (Single Instruction stream, Multiple Data stream, SIMD). Несколькими потоками данных являются последовательности элементов данных, с которыми работают разные процессоры, каждый в своей памяти. Третья схема функционирования системы предполагает использование нескольких независимых процессоров,

которые выполняют разные программы и работают с разными последовательностями данных. Системы, функционирующие по этой схеме, называются системами *с множественным потоком команд и множественным потоком данных* (Multiple Instruction stream, Multiple Data stream, MIMD). Системы четвертого типа известны как системы *с множественным потоком команд и одиночным потоком данных* (Multiple Instruction stream, Single Data stream, MISD). В таких системах единый поток данных обрабатывается несколькими процессорами, выполняющими разные программы. Эта форма параллельных вычислений редко используется на практике, и поэтому мы не будем ее рассматривать.

Основное внимание в данной главе уделяется структурам MIMD, имеющим наиболее широкое применение. Однако сначала мы коротко рассмотрим структуру SIMD и покажем, для каких приложений она используется.

12.2. Матричная обработка данных

Первым способом параллельной обработки данных, который был изучен и реализован на практике, была архитектура SIMD, называемая также *матричной обработкой* (array processing). В начале 1970-х годов в университете штата Иллинойс на основе принципа матричной обработки была создана система ILLIAC-IV, позднее изготовленная компанией Burroughs Corporation.

Структура матричного процессора показана на рис. 12.1. Двухмерный массив процессорных элементов обрабатывает поток команд, получаемых от центрального управляющего процессора. Каждая из команд выполняется всеми процессорными элементами одновременно. Каждый процессорный элемент обменивается данными со своим ближайшим соседом. Возможны и обходные соединения, но на рисунке они не приведены.

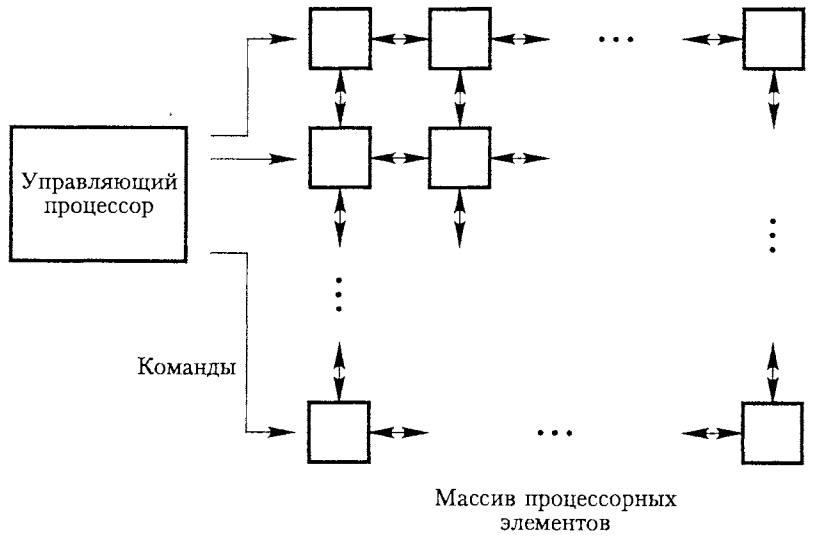


Рис. 12.1. Матричный процессор

Массив процессорных элементов может использоваться для обработки двухмерных данных. Например, если каждый элемент массива определяет точку в пространстве, то массив может служить для вычисления значений температуры внутри плоской теплопроводной поверхности. Предположим, что температура на ее краях постоянна. Приближенное решение в дискретных точках, представленных процессорными элементами, формируется так. В начальный момент времени процессорные элементы, расположенные на границе области, инициализируются некоторыми заданными значениями температуры, а все внутренние точки — произвольными, не обязательно одинаковыми, значениями. Затем все элементы начинают параллельно оценивать значения температуры в соответствующих точках как среднее арифметическое значений температуры в четырех соседних точках. Вычисления повторяются до тех пор, пока разность результатов последовательных вычислений не окажется меньше некоторого указанного значения.

Для выполнения таких расчетов необходимо, чтобы каждый элемент матричного процессора обеспечивал обмен данными с соседними элементами через показанные на рисунке соединения. Каждый элемент должен иметь несколько регистров и небольшой объем локальной памяти для хранения данных. Кроме того, важно наличие у элемента так называемого сетевого регистра, служащего для обмена данными с соседними элементами. Центральный процессор направляет всем элементам команду пересылки значений сетевых регистров на один уровень вверх, вниз, вправо или влево. Каждый элемент должен содержать АЛУ для выполнения арифметических команд, получаемых от центрального процессора. Организовав итеративный цикл, с помощью этих базовых средств команды элементам можно передавать многократно. При этом необходимо, чтобы управляющий процессор определял момент, когда каждый процессорный элемент вычислит температуру в определенной точке с указанной точностью. С этой целью по достижении заданного значения внутренний бит состояния каждого из элементов должен устанавливаться в 1. Соединения между элементами позволяют контроллеру устанавливать все биты состояния в 1, что равнозначно завершению операции.

При разработке матричных процессоров возникает вопрос: что эффективнее использовать — несколько мощных процессоров или большое количество очень простых процессоров. Примером реализации первого подхода может служить суперкомпьютер ILLIAC-IV. В его состав входят 64 процессора, обрабатывающих 46-разрядные числа. Примером реализации второго подхода является применение матричных процессоров, разработанных в конце 1980-х годов. В системе CM-2 производства корпорации Thinking Machines можно было установить до 65536 процессоров, но разрядность каждого из них равнялась одному биту. В системе Maspar MP-1216 допускалась установка 16384 процессоров разрядностью 4. Системы Cambridge Parallel Processing Gamma II Plus могли содержать до 4096 процессоров, обрабатывающих данные длиной в 1 байт или 1 бит. Разработчики этих систем полагали, что в SIMD-архитектуре наличие высокого уровня параллелизма эффективнее использования небольшого количества мощных процессоров.

Сфера применения матричных процессоров является достаточно узкой. В первую очередь они предназначены для решения вычислительных задач, связанных с обработкой матриц и векторов. Напомним, что для решения подобных задач

подходят и суперкомпьютеры с векторной архитектурой. Главным отличием матричных процессоров от векторных систем является то, что при работе со вторыми высокая производительность достигается за счет интенсивной конвейеризации, а при использовании первых — за счет максимальной степени параллелизма, являющегося результатом параллельной работы компьютерных модулей. Но ни матричные, ни векторные компьютеры не могут значительно ускорить обычные вычисления, поэтому они не пользуются коммерческим успехом.

12.3. Архитектура мультипроцессорных систем общего назначения

Описанные в предыдущем разделе матричные системы предназначены для выполнения вычислений с ярко выраженным параллелизмом данных. Для других задач, где нет столь явно выраженного параллелизма данных, гораздо лучше подходит архитектура MIMD, в которой множество процессоров могут независимо и параллельно выполнять разные подпрограммы.

На рис. 12.2–12.4 продемонстрированы три возможных способа реализации мультипроцессорной системы типа MIMD. Самая простая система представлена на рис. 12.2. Она состоит из n процессоров, k модулей памяти и *коммуникационной сети*, связывающей процессоры и память. Сеть может стать причиной значительной задержки при обращении процессора к памяти. Система, в которой такая задержка одинакова для всех операций доступа к памяти, называется *мультипроцессорной системой с однородным доступом к общей памяти* (Uniform Memory Access, UMA) или *системой с общей памятью*. Поскольку процессоры выполняют команды с огромной скоростью, слишком большие задержки на выборку из памяти команд и данных для них не приемлемы. Однако коммуникационные сети с малым временем задержки имеют очень сложную структуру и высокую стоимость.

Достичь высокого быстродействия всех процессоров можно путем непосредственного соединения с ними модулей памяти. Архитектура такой системы показана на рис. 12.3. Каждый процессор имеет доступ не только к собственной локальной памяти, но и к памяти других процессоров сети. Но поскольку при обращении к памяти других процессоров запросы проходят через сеть, они выполняются дольше, чем обращения к локальной памяти. Системы этого типа называются *мультипроцессорными системами с неоднородным доступом к памяти* (Non-Uniform Memory Access, NUMA).

В схемах, приведенных на рис. 12.2 и 12.3, используется *глобальная память*, к каждому из модулей которой может обратиться любой из процессоров. На рис. 12.4 демонстрируется схема иной организации системы, характеризующаяся тем, что все модули памяти являются собственностью непосредственно соединенных с ними процессоров. Ни один из процессоров не может обратиться к удаленной памяти без взаимодействия с удаленным процессором, которому она принадлежит. Взаимодействие между этими двумя процессорами осуществляется в форме обмена сообщениями. Системы такого типа называются системами с *распределенной памятью и высокоскоростным протоколом передачи сообщений.*

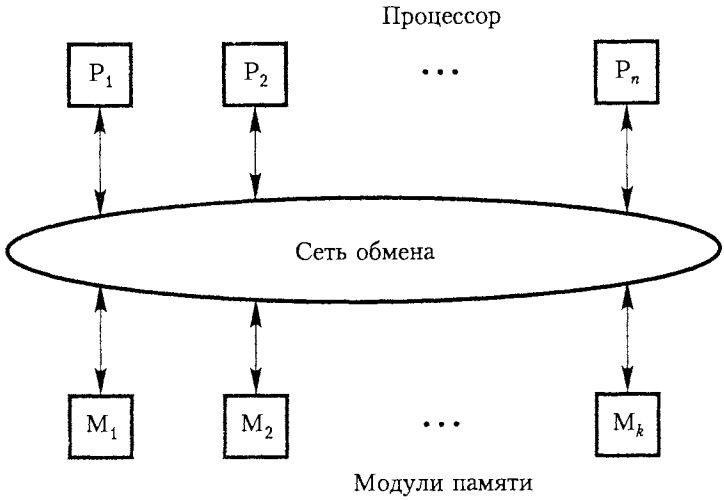


Рис. 12.2. Мультипроцессорная система типа UMA

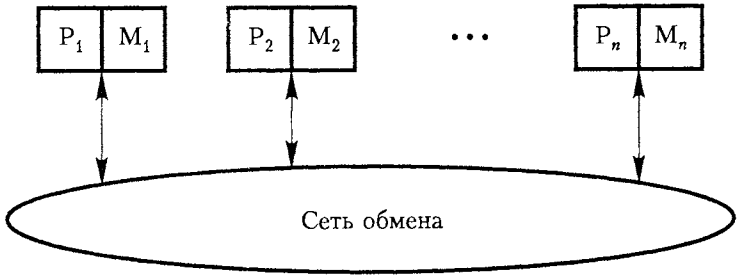


Рис. 12.3. Мультипроцессорная система типа NUMA

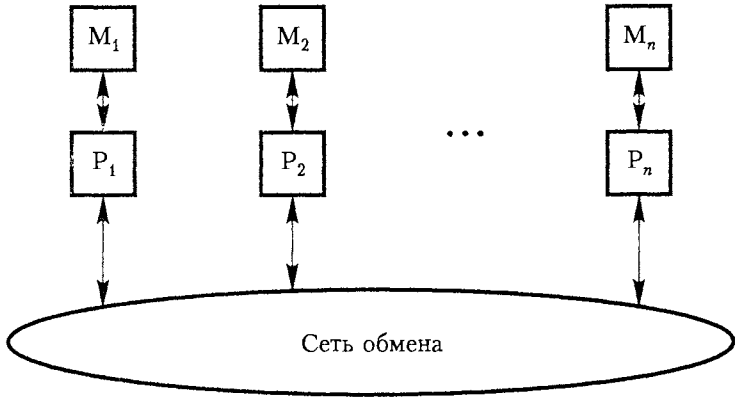


Рис. 12.4. Мультипроцессорная система с распределенной памятью

До сих пор в этом разделе в качестве основных функциональных устройств мультипроцессорной системы рассматривались процессоры и модули памяти. Мы не обсуждали модули ввода-вывода, хотя любая мультипроцессорная система обязательно должна иметь эффективные инструменты такого рода. Существуют разные средства, предназначенные для ввода и вывода информации. Отдельные модули ввода-вывода могут быть соединены прямо с коммуникационной сетью, обеспечивая стандартные интерфейсы ввода-вывода, о которых рассказывалось в главе 4. Кроме того, некоторые функции ввода-вывода могут быть интегрированы прямо в процессорные модули.

Высокоуровневое представление возможных способов организации мультипроцессоров показано на рис. 12.2–12.4. Производительность и стоимость подобных систем в значительной степени зависит от того, как реализованы их схемы. В следующих двух разделах речь пойдет о наиболее популярных схемах организации сетей обмена и о структуре иерархии памяти.

12.4. Коммуникационные сети

А сейчас мы рассмотрим возможные способы реализации коммуникационных сетей в мультипроцессорных системах. Коммуникационная сеть должна обеспечивать передачу данных между любыми двумя модулями системы. Кроме того, сеть может использоваться для широковещательной передачи информации от одного модуля системы ко многим другим. Трафик в сети формируется из запросов (например, на чтение или запись), пересылаемых данных и различных команд.

Каждая конкретная сеть характеризуется такими параметрами, как стоимость, полоса пропускания, реальная скорость передачи данных и простота реализации. Термин *полоса пропускания* обозначает пропускную способность соединений и определяется как количество битов или байтов данных, которые могут проходить через это соединение за единицу времени. *Реальная скорость передачи данных* соответствует реальному объему данных, проходящему через соединение за единицу времени. Она меньше полосы пропускания, поскольку при передаче данных могут возникать паузы.

Информация по сети передается, как правило, в виде *пакетов* фиксированной длины и фиксированного формата. Например, запрос на чтение может представлять собой один пакет, содержащий адрес источника (процессорного модуля) и места назначения (модуля памяти), а также поле команды, где указан тип операции чтения, которую требуется выполнить. Запрос на запись одного слова в модуль памяти тоже может состоять из одного пакета, содержащего записываемые данные. А вот ответ на запрос чтения, включающий целый блок кэш-памяти, может пересылаться в виде нескольких последовательных пакетов. Несколько пакетов может потребоваться и для длинных сообщений.

В идеале полный пакет должен пересылаться между любыми двумя узлами сети за один такт в параллельном режиме. Для этого нужны соединения, состоящие из большого количества проводов. Однако в целях упрощения архитектуры и снижения стоимости сети соединения обычно делают более узкими, а пакеты разделяют на части, каждая из которых может быть переслана за один такт.

12.4.1. Общая шина

Простейший и самый экономичный способ объединения большого количества модулей в единую систему заключается в использовании общей шины. Эта архитектура подробно обсуждалась в главе 4; все сказанное там применимо и к мультипроцессорным системам. Поскольку несколько модулей соединены с одной шиной и каждому из них в любой момент может быть направлен запрос на передачу данных, необходима эффективная арбитражная схема. Примеры таких схем также приводились в главе 4.

Самым простым режимом функционирования системы с общей шиной является режим, при котором шина предоставляется конкретной паре компонентов на все время операции передачи данных. Например в случае, когда процессор выдает запрос на чтение, он занимает шину до тех пор, пока не получит из памяти нужные данные. Поскольку модулю памяти для доступа к данным требуется некоторое время (см. главу 5), шина простаивает с момента получения запроса до тех пор, пока память не будет готова вернуть данные. Готовые данные направляются процессору, и когда процесс передачи завершается, шина может быть выделена для выполнения другого запроса.

Существует схема под названием *протокол с разделением транзакций*, позволяющая использовать время простоя шины для обработки другого запроса. Рассмотрим следующую процедуру обработки последовательности запросов на чтение, поступающих от разных процессоров. После передачи адреса, указанного в первом запросе, шина может быть переназначена для передачи адреса, заданного в другом запросе. В случае, если данный запрос адресован другому модулю памяти, с этого момента два модуля памяти могут обрабатывать полученные запросы параллельно. Если ни один из модулей еще не завершил операцию обращения к памяти, шина может быть выделена третьему запросу и т. д. Когда первый модуль наконец завершит цикл обращения к памяти, он воспользуется шиной для передачи слова запросившему его компоненту. Обратите внимание, что фактическое время между выдачей адреса и получением слова не является критически важным параметром. Операции передачи адреса и данных для различных запросов выполняются независимо и могут чередоваться друг с другом.

Возможность более эффективного использования полосы пропускания шины обеспечивает протокол с разделением транзакций. Достигаемая с его помощью производительность зависит от соотношения времени доступа к памяти и времени передачи данных по шине. Производительность можно повысить за счет усложнения архитектуры шины. А необходимость в ее усложнении вызвана двумя причинами. Во-первых, поскольку модулю памяти нужно знать, от какого источника поступил данный запрос, к запросу должен присоединяться тег идентификации источника. После выполнения запроса этот тег используется для отправки источнику требуемых данных. А во-вторых, не только процессор, но и все остальные модули должны иметь возможность выступать в роли хозяев шины.

Мультипроцессорные системы, в которых применяется шина с разделением транзакций, содержат от 4 до 32 процессоров. Возможность подключения еще большего количества процессоров ограничена полосой пропускания шины. Для увеличения полосы пропускания можно увеличить количество составляющих

шину линий. В большинстве операций передачи данных между процессором и памятью по шине пересылаются блоки кэша, состоящие из некоторого количества слов. Если шина достаточно широка, чтобы по ней можно было одновременно передавать несколько слов, блок пересылается быстрее. Так, в микропроцессоре Challenge производства Silicon Graphics Corporation используется шина, по которой могут одновременно передаваться 256 бит данных.

Основным недостатком систем с общей шиной является то, что к шине можно подключить не так уж много модулей. Обычная шина хорошо справляется с 10–15 подключенными к ней модулями. При расширении шины количество подключенных модулей может быть удвоено. Однако оно все равно ограниченно, поскольку при очень большом числе модулей шина независимо от полосы пропускания уже просто не будет успевать обслуживать все их запросы. Кроме того, с повышением электрической нагрузки из-за большого количества подключенных к шине устройств увеличивается задержка на распространение сигнала по ней. Значительное повышение скорости передачи данных достигается в сетях, поддерживающих параллельное выполнение нескольких операций передачи данных.

12.4.2. Сети с координатной коммутацией

На рис. 12.5 показана сеть с коммутируемыми соединениями. Эта схема, разработанная для телефонных сетей, называется *координатным коммутатором* (crossbar switch). Для наглядности все ключи на рисунке изображены в виде механических выключателей, но на самом деле это, конечно же, электронные ключи. В приведенной схеме любой модуль, Q_i, может быть соединен с другим модулем, Q_j, путем замыкания соответствующего ключа. Сети, в которых существует прямое соединение между любой парой узлов, именуются *полносвязными*. В полносвязной сети может выполняться множество параллельных операций передачи. Если n узлов должны переслать данные n другим узлам, все эти операции могут быть выполнены одновременно. Сеть, в которой операции передачи никогда не блокируются из-за отсутствия свободного пути, называется *сетью без блокировок*.

На рис. 12.5 в каждой точке пересечения соединений показан единственный ключ. Однако в реальном мультипроцессоре используются более сложные соединения, и поэтому в каждой точке располагается целый набор ключей. В сети, соединяющей n модулей, число точек пересечения равняется n^2, поэтому с увеличением количества модулей возрастает и общее число ключей. В результате сеть получается громоздкой и дорогостоящей. По этой причине координатная коммутация обычно применяется для сетей с небольшим количеством узлов.

Один из самых больших координатных коммутаторов используется в системе Sun E10000, в которой 16 четырехпроцессорных узлов соединены координатным коммутатором 16 × 16. Существует возможность реализовать и многоуровневый коммутатор, состоящий из нескольких соединенных друг с другом координатных коммутаторов. Таким способом можно соединить и большее количество процессоров. Подобные схемы используются в системах Fujitsu VPP5000, Hitachi SR8000 и NEC SX-5. Благодаря своей высокой производительности многоуровневые координатные коммутаторы приобретают все большую популярность.

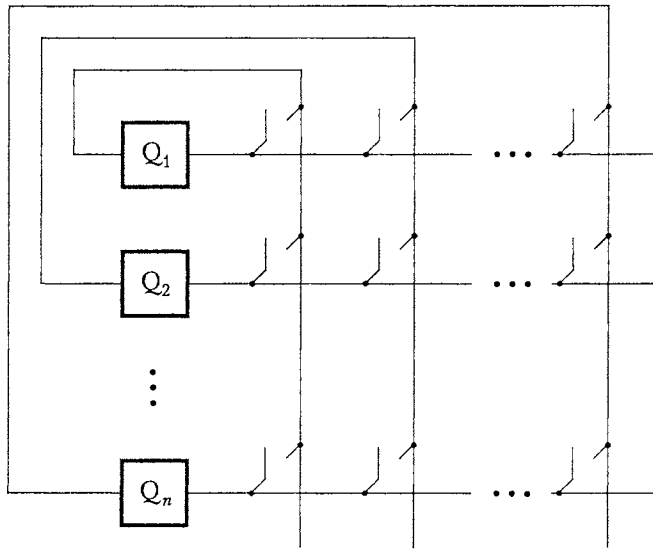


Рис. 12.5. Сеть с координатной коммутацией

12.4.3. Многоступенчатые сети

В описанных выше системах с общей шиной и координатной коммутацией используется одноступенчатое соединение между любыми двумя модулями. Можно создать такую сеть обмена, в которой соединение между двумя узлами будет формироваться с помощью нескольких ступеней ключей. На рис. 12.6 показана трехступенчатая сеть, соединяющая восемь модулей. Из-за особенностей структуры соединений ее элементов эта сеть получила название коммутационной сети с перетасовкой (*shuffle*) — по аналогии с перетасовкой игральных карт путем разделения колоды карт на две части и последующего их объединения с чередованием карт из обеих частей.

Каждый прямоугольник на рис. 12.6 представляет собой переключатель 2×2, который может соединить любой из входов с любым из выходов. Кроме того, если оба входа должны быть соединены с разными выходами, он может соединить их как прямо, так и накрест. Если два входа запросили один и тот же выход, может быть удовлетворен только один запрос. Второй запрос блокируется, пока переключатель не освободится. Можно показать, что сеть с s ступенями может использоваться для соединения 2^s модулей. В этом случае существует только один путь от модуля Q_i к модулю Q_j. Таким образом, сеть обеспечивает соединение между двумя произвольными модулями. Однако она не в состоянии одновременно обслуживать большое количество запросов. Например, соединение между точками Q_0 и Q_4 не может быть установлено одновременно с соединением между точками Q_1 и Q_5.

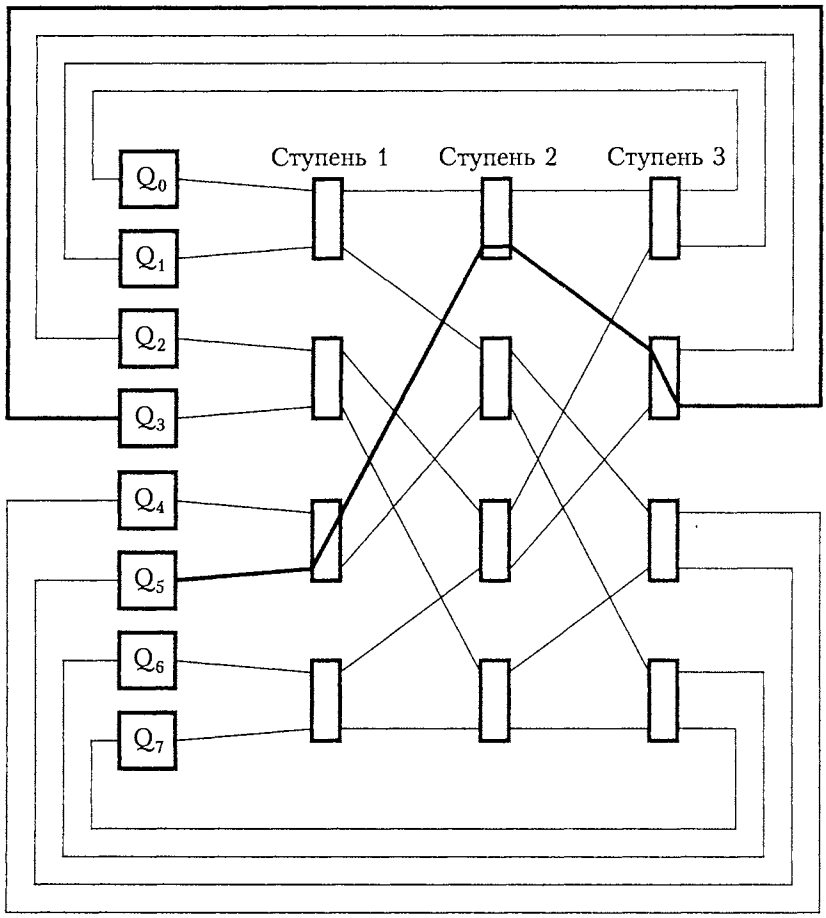


Рис. 12.6. Многоступенчатая сеть типа shuffle

Многоступенчатая сеть дешевле сети с координатной коммутацией. Для соединения n узлов по схеме, показанной на рис. 12.6, необходимо s = $\log_2 n$ ступеней с $n/2$ переключателями на каждой. Поскольку каждый переключатель состоит из четырех ключей, общее их количество равняется

$$4 \times \frac{n}{2} \times \log_2 n = 2n \times \log_2 n$$

что для больших сетей значительно меньше, чем n^2 ключей, необходимых для соединения модулей в сети с координатной коммутацией.

Передача запроса по сети выполняется следующим образом. Источник направляет в сеть двоичную последовательность, представляющую адрес приемника. Эта последовательность передается по сети, и на каждой ступени анализируется очередной ее бит, который определяет состояние переключателя. В нашем примере на ступени 1 используется старший бит, на ступени 2 — средний бит, а на

ступени 3 — последний, младший бит. Когда запрос поступает на один из входов переключателя, он маршрутизируется к верхнему выходу, если управляющий бит имеет значение 0, или к нижнему выходу в противном случае. Обратите внимание на пример, приведенный на рис. 12.6, где жирной линией показан маршрут следования по сети запроса от источника Q_5 к приемнику Q_3. Этот маршрут определяется двоичной последовательностью 011, составляющей адрес приемника.

Хорошим примером мультипроцессора на основе многоступенчатой сети может служить система BBN Butterfly, которая в свое время выпускалась компанией BBN Advanced Computers. Эта 46-процессорная система включала трехступенчатую сеть на базе переключателей 4 × 4. Для маршрутизации запросов на каждой ступени переключателей были задействованы последовательные 2-битовые поля адреса приемника. Еще одним примером является мультипроцессор IBM RS/6000 SP, в котором для соединения кластеров процессоров может использоваться многоступенчатая сеть.

По количеству параллельных соединений многоступенчатые сети уступают сетям с координатной коммутацией, но зато их реализация дешевле. Интерес к этим сетям достиг своего пика в 1980-е годы, но за последние несколько лет значительно уменьшился. Популярность получили другие схемы, о которых рассказывается далее.

12.4.4. Сети с топологией гиперкуба

Во всех трех описанных выше коммутационных схемах передача запросов между двумя модулями происходит с некоторой задержкой. Эти схемы могут использоваться для реализации мультипроцессоров типа UMA. Далее мы рассмотрим топологии, подходящие только для мультипроцессоров типа NUMA. Первая и наиболее популярная сеть этого типа, соединяющая 2^n узлов, имеет топологию n-мерного куба, называемую *гиперкубом* (hypercube). Помимо коммуникационных схем каждый узел обычно содержит процессор и модуль памяти, а также некоторые средства ввода-вывода.

На рис. 12.7 изображен трехмерный гиперкуб. Здесь кружочками обозначены коммуникационные схемы узлов. Соединяемые узлами функциональные устройства на рисунке не показаны. Ребра куба представляют собой двунаправленные связи между соседними узлами. В n-мерном гиперкубе каждый узел непосредственно связан с n соседними узлами. Двоичные адреса узлам удобнее назначать таким образом, чтобы адреса двух соседних узлов отличались единственным битом, как на рис. 12.7.

Маршрутизация сообщений в гиперкубе выполняется очень просто. Передача сообщения от процессора в узле N_i процессору узла N_j происходит следующим образом. Сначала двоичные адреса источника i и приемника j сравниваются, начиная с младшего бита и заканчивая старшим. Предположим, что они отличаются разрядом p. Далее узел N_i передает сообщение своему соседу, адрес которого, k, отличается от i в разряде p. Узел N_k пересылает сообщение своему соседу, используя ту же схему сравнения адресов. После каждой передачи от одного узла к другому сообщение приближается к узлу-приемнику N_j. Например, для передачи

сообщения от узла N_2 к узлу N_5 требуются 3 пересылки, осуществляемые через узлы N_3 и N_1. Причем именно это количество пересылок соответствует максимальному расстоянию, проходимому сообщением в n-мерном гиперкубе.

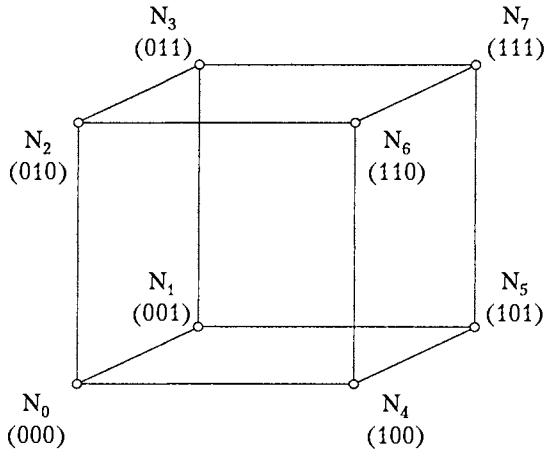


Рис. 12.7. Сеть в виде трехмерного гиперкуба

Сканирование адресов назначения в направлении справа налево не является единственным методом маршрутизации сообщений. Может использоваться любая другая схема, при которой с каждой пересылкой от узла к узлу сообщение приближается к месту назначения, а для дальнейшей маршрутизации на каждом узле используется только локальная информация. Если кратчайший путь передачи сообщения не доступен, оно может быть направлено по более длинному пути. Однако при этом следует избегать зацикливания, когда сообщение проходит по замкнутому маршруту и возвращается к исходной точке, так и не достигнув места назначения.

Сети с топологией гиперкуба используются во множестве систем. Наиболее известными среди них являются Intel iPSC, в которой семимерный куб соединяет до 128 узлов, и NCUBE NCUBE/ten, где 1024 узла соединены десятимерным кубом. В начале 1990-х годов популярность сетей с топологией гиперкуба значительно снизилась, поскольку их стали вытеснять более привлекательные структуры — ячеистые сети (о них рассказывается в следующем разделе).

12.4.5. Сети с ячеистой топологией

Одним из самых удобных способов соединения большого количества узлов является *ячеистая сеть* (mesh network). Пример ячеистой сети с 16 узлами приведен на рис. 12.8. Соединения между ее узлами являются двунаправленными. Такие сети завоевали популярность в начале 1990-х годов и стали быстро вытеснять гиперкубические структуры в архитектуре крупных мультипроцессорных систем.

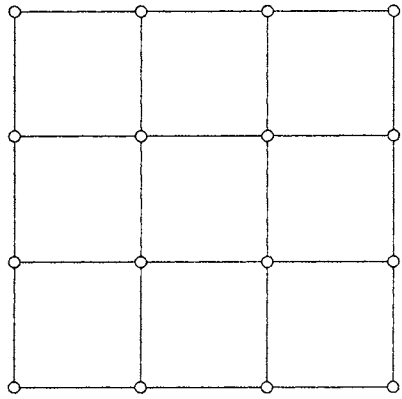


Рис. 12.8. Двухмерная ячеистая сеть

Маршрутизация в ячеистой сети может осуществляться несколькими способами. Один из простейших и наиболее эффективных способов выбора пути между источником N_i и приемником N_j заключается в том, что сначала выполняется передача данных в горизонтальном направлении от узла N_i к узлу N_j. По достижении столбца, в котором расположен узел N_j, пересылка производится по вертикали. Широко известными примерами мультипроцессорных систем с ячеистой топологией являются компьютер Paragon компании Intel и экспериментальные системы Dash и Flash, разработанные в Стэндфордском университете, а также система Alewife, созданная в Массачуссетском технологическом институте.

Если соединить между собой узлы, находящиеся на противоположных сторонах (рис. 12.8), получится сеть, состоящая из набора двунаправленных горизонтальных колец, соединенных с такими же вертикальными кольцами. В подобных сетях, то есть имеющих тороидальную топологию (torus), значительно сокращается среднее время передачи информации, правда, они сложнее и дороже. Сеть такого типа используется в системах AP3000 компании Fujitsu.

Как обычную, так и тороидальную ячеистую схему можно реализовать в виде трехмерной сети с соединениями между соседними узлами в направлениях *X*, *Y* и *Z*. Трехмерная тороидальная сеть используется, в частности, в мультипроцессоре Cray T3E.

12.4.6. Сети с древовидной топологией

Еще одной популярной топологией сети обмена является иерархически структурированная сеть в виде дерева. На рис. 12.9, *а* показано дерево, соединяющее 16 модулей. Каждый родительский узел этого дерева позволяет соединять по два дочерних узла за раз. Узлы промежуточных уровней, такие как узел A, могут соединять один из дочерних узлов с родительским. Таким образом осуществляется взаимодействие между двумя удаленными узлами. Через один узел дерева не может одновременно проходить несколько соединений.

Древовидная сеть хорошо подходит для случаев, когда через корневой узел сети проходит небольшой трафик. Если же трафик возрастает, производительность сети резко снижается, поскольку корневой узел становится ее узким местом. Для

повышения пропускной способности сети с древовидной структурой можно увеличить количество соединений на верхних уровнях иерархии. В результате получается *толстое дерево* (fat tree), где каждый узел имеет более одного родительского узла. Пример сети с топологией толстого дерева показан на рис. 12.9, *б*. Здесь каждый узел имеет два родительских узла. Подобная сеть использовалась в системе CM-5 производства Thinking Machines Corporation.

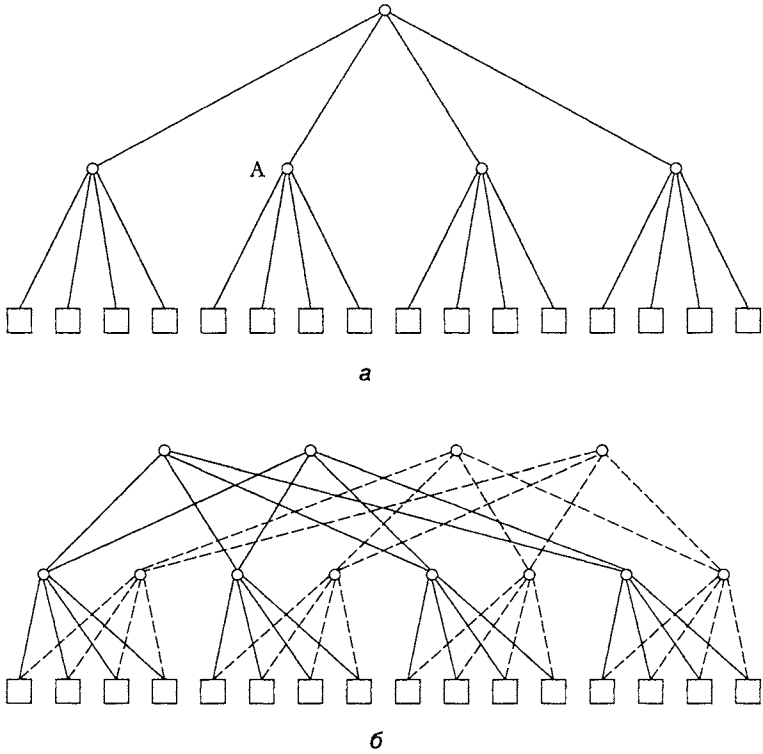


Рис. 12.9. Древовидные сети: дерево с четырьмя ветвями (*а*); толстое дерево (*б*)

12.4.7. Сеть с кольцевой топологией

Одной из простейших сетевых топологий является кольцевое соединение всех узлов системы (рис. 12.10, *а*). Главное преимущество такой структуры состоит в простоте ее реализации. Соединения в кольце могут быть достаточно широкими, благодаря чему по ним можно параллельно передавать целый пакет. А вот слишком длинные кольца не эффективны, поскольку среднее время передачи информации между двумя узлами получается слишком большим.

Сети с кольцевой топологией могут служить строительными блоками в топологиях, описанных в предыдущих разделах, в частности в ячеистых, древовидных сетях и гиперкубах. Рассмотрим такой простой пример. При использовании

колец в древовидной структуре получается иерархия, показанная на рис. 12.10, *б*. На этом рисунке изображена двухуровневая иерархия, но возможно и большее количество уровней. В случае использования нескольких маленьких колец вместо одного большого уменьшается задержка при передаче сообщения в пределах кольца, а также задержка во время передачи данных между соседними кольцами. Недостатком этой схемы является то, что кольца верхних уровней нередко становятся узким местом сети, задерживающим значительную часть трафика.

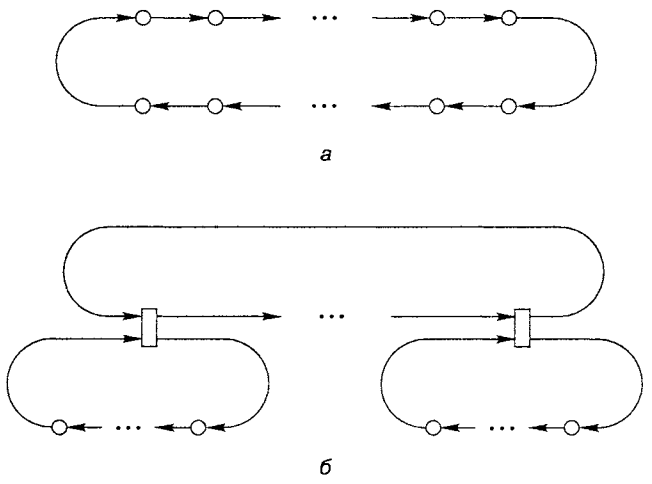


Рис. 12.10. Сеть с кольцевой топологией: одно кольцо (*а*); иерархия колец (*б*)

Коммерческими компьютерами, в которых используются сети с кольцевой топологией, являются Exemplar V2600 производства Hewlett-Packard и KSR-2 производства Kendal Square Research. Кольцевая топология применялась также в экспериментальных системах Hector и NUMAchine в университете Торонто.

12.4.8. Практические рекомендации

Выше было рассмотрено несколько топологий, используемых для реализации коммуникационных сетей в мультипроцессорных системах. Трудно доказать, что какая-то конкретная топология значительно лучше остальных. У каждой из них есть свои преимущества и недостатки. При выборе топологии необходимо учитывать следующее.

Основным требованием к работе коммуникационной сети является ее быстродействие: сеть должна быть сравнительно быстрой и обладать достаточной пропускной способностью. Для этого необходима высокая скорость передачи данных по коммуникационным соединениям и простой механизм маршрутизации, позволяющий быстро принимать решения о дальнейшем маршруте следования запросов. Сеть должна быть легкореализуемой, а соединения — простыми, чтобы пакеты имели простую структуру. Усложнение сети немедленно отразится на ее стоимости, которая является еще одним очень важным критерием.

Для разных задач нужны мультипроцессорные системы разного масштаба (то есть с разным количеством процессорных элементов). Идеальной структурой сети считается такая, при которой возможна работа любого количества процессоров, от нескольких единиц до тысяч. Способность мультипроцессорной системы повышать свою производительность при наращивании вычислительных мощностей называется *масштабируемостью*. Вы можете за небольшую плату приобрести систему малой производительности, а затем по мере необходимости наращивать ее вычислительные мощности. К сожалению, для многих коммерческих продуктов это невозможно. Как правило, начальная стоимость даже небольшой системы очень высока, поскольку уже первоначальная конфигурация такой системы должна включать значительную часть коммуникационного аппаратного обеспечения, необходимого для ее функционирования.

Наряду с базовыми средствами коммуникации между источником и приемником запросов желательно иметь возможность *широковещательной* (broadcasting) рассылки сообщений, при которой одно сообщение передается по всей сети и принимается всеми ее узлами. Удобно также, если поддерживается рассылка сообщений подмножеству узлов сети. Такая рассылка часто называется *групповой* (multicasting).

От выбранной топологии коммуникационной сети зависит реализация схем, используемых для согласования копий одних и тех же данных в кэш-памяти разных процессоров. Об этих схемах рассказывается в разделе 12.6.2.

Еще одним важным фактором, который должен учитываться при выборе архитектуры коммуникационной сети, является ее надежность. Чем сложнее сеть, тем больше вероятность различных сбоев. В идеале система должна продолжать функционировать, даже если одно из ее соединений выйдет из строя. Для этого нужно, чтобы в сети было как минимум два пути между каждой парой узлов. В общем случае простые сети считаются более надежными и отказывают не чаще, чем процессорные модули и модули памяти. При условии вложения необходимых средств можно разработать высоконадежные сети, содержащие дополнительное аппаратное обеспечение. Однако эта тема выходит за рамки нашей книги.

Для того чтобы показать, как оцениваются все эти характеристики, давайте проведем небольшое сравнение сетей ячеистой и кольцевой структуры.

Ячеистые и кольцевые сети

Сети ячеистой и кольцевой топологии характеризуются высоким быстродействием соединений «точка-точка» (то есть соединений между двумя узлами). Они могут работать в маленьких конфигурациях и легко расширяются. При этом постепенное наращивание проще выполнять в кольцевой сети.

На рис. 12.8 и 12.10 узлы сетей были обозначены кружочками, а соединения — одинарными линиями. Давайте рассмотрим схему более подробно. На рис. 12.11 показаны коммуникационные соединения одного узла, к которому подключен процессорный модуль. Переключающий блок содержит схему для выбора пути передачи и буферы для хранения передаваемых данных. За один такт данные пересылаются из буфера одного узла в буфер другого. На рис. 12.11, *а* изображен узел двухмерной ячеистой сети. Поскольку и по горизонтали, и по вертикали необходимы

двунаправленные соединения, к узлу должно подходить восемь разных соединений. Ширина каждого из них ограничена общим количеством линий, которые могут использоваться в схеме (с учетом ее стоимости и плотности размещения элементов на ней). Поэтому маловероятно, что каждое отдельное соединение будет достаточно широким для передачи всего пакета. Это означает, что пакет следует разделить на части, соответствующие ширине соединения. Для обозначения части пакета, которая может быть принята схемой узла для последующей передачи или буферизации в случае, если дальнейший путь заблокирован другой передачей, используется термин *флит* (flit, FLow control digIT— символ управления потоком).

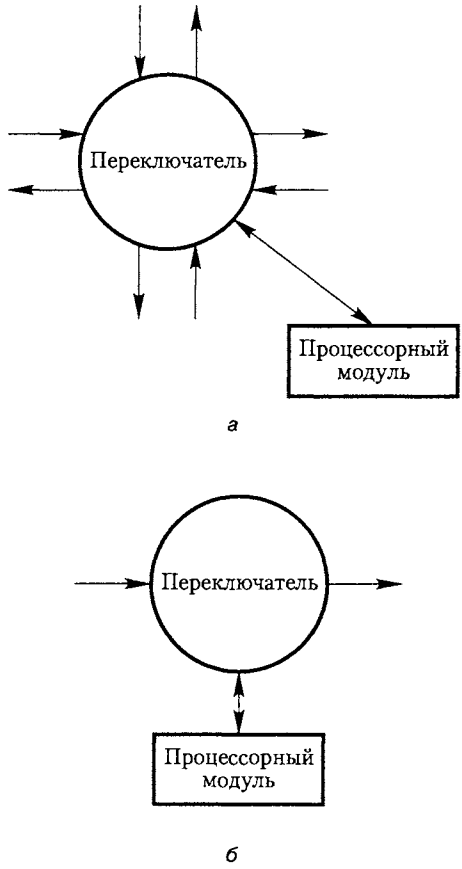


Рис. 12.11. Узлы в различных сетях: узел ячеистой сети (*а*); узел кольцевой сети (*б*)

Как же передавать по сети пакет, разделенный на несколько флитов? Простейшая схема, называемая маршрутизацией *с промежуточным хранением* (store and forward), предполагает использование на каждом узле достаточно большого буфера, в котором помещались бы все флиты пакета. В этом случае весь пакет может быть

передан по частям от одного узла к другому, где он будет храниться до передачи следующему узлу. (При этом количество тактов, необходимых для передачи пакета между двумя узлами, зависит от числа флитов.) Недостатком этой схемы является увеличение размера буферов и задержки на передачу пакета от узла к узлу. Альтернативой подобному способу служит маршрутизация путем *коммутации каналов* (wormhole — буквально: «червячный ход»), называемая также *конвейерной* маршрутизацией. При коммутации каналов каждый пакет может рассматриваться как *«червь»*, ползущий по сети. Его первый флит содержит *заголовок*, включающий адрес места назначения. Перемещаясь по сети, этот флит прокладывает путь, по которому будут передаваться остальные флиты пакета. Хвост «червя» закрывает проложенный путь. Голова «червя» может быть временно заблокирована в любом из узлов, если через него проползает другой «червь». Но как только голова «червя» продолжает движение, вслед за ней на последующих тактах по тому же пути проходят остальные его части. Для остановки всех флитов пакета при блокировке первого из них нужен какой-нибудь управляющий механизм; простейший механизм заключается в использовании двух буферов в каждом узле, соответствующих двум направлениям передачи. Маршрутизация с коммутацией каналов характеризуется меньшей задержкой, чем маршрутизация с промежуточным хранением, поскольку первый флит пакета следует по своему маршруту, не дожидаясь остальных.

Коммутация каналов является частным случаем технологии под названием *коммутация цепей*, применяемой в телефонных сетях, где соединение (то есть путь в сети) устанавливается при наборе номера. Разговор осуществляется через это соединение, называемое *цепью* или *линией*. Когда один из говорящих вешает трубку, цепь деактивизируется. В случае коммутации каналов соединение устанавливается первым флитом. Его движение может быть временно заблокировано другим пакетом, как описано выше. Однако после установки соединения оставшиеся части пакета перемещаются в направлении места назначения без задержек. Технология, основанная на противоположном принципе, при котором весь пакет буферизируется в каждом узле, как и при использовании метода с промежуточным хранением, называется *коммутацией пакетов*. В этом случае соединение не устанавливается и пакет перемещается по сети по мере доступности буферов в каждом из узлов на пути его следования.

На рис. 12.11, *б* показаны соединения с узлом кольцевой сети. В этой сети передача выполняется только в одном направлении. Поэтому при том же количестве проводов ширина линии может быть вчетверо больше, чем в ячеистой сети. Это означает, что весь пакет может быть передан от одного узла к другому за один такт. Для иерархической кольцевой сети (рис. 12.11, *б*) показан узел кольца самого нижнего уровня, с которым соединен процессорный модуль. Межкольцевые интерфейсы должны иметь по два входных и два выходных соединения — пару для кольца верхнего уровня и пару для кольца нижнего уровня.

Маршрутизация в иерархической кольцевой сети выполняется просто. Пакет никогда не блокируется, разве что в межкольцевом интерфейсе, если в нем столкнутся входящие пакеты, следующие сверху вниз и снизу вверх. Для обработки такой ситуации интерфейс должен содержать два буфера (очереди) — по одному

для каждого из этих двух пакетов. Процессорный модуль может выпустить в кольцо новый пакет в любой момент, когда его узел не занят маршрутизацией пакета от соседнего узла.

Теперь рассмотрим вопрос широковещательной и групповой рассылки данных. Рассылка такого типа характерна для кольцевых сетей. Например, для широковещательной рассылки в иерархической кольцевой сети пакет нужно выслать из кольца верхнего уровня. Во время «путешествия» этого пакета по сети на каждом межкольцевом интерфейсе будет создаваться и передаваться в кольца нижнего уровня его копия. Процесс будет повторяться на всех уровнях, так что в результате копии исходного пакета окажутся разосланными по всем узлам колец самого нижнего уровня. В ячеистой сети широковещательную рассылку пакета организовать труднее, поскольку пакет должен быть разбит на флиты и его движение во многих узлах может блокироваться другим трафиком. Более того, трудно определить момент завершения процесса широковещательной рассылки.

Главным недостатком иерархической кольцевой сети является то, что при большом количестве пересылаемых пакетов кольцо на вершине иерархии может стать ее узким местом. Фиксированная полоса пропускания кольца верхнего уровня ограничивает масштабируемость этой архитектуры несколькими сотнями процессоров, тогда как ячеистые сети могут содержать тысячи процессоров.

Все сказанное выше свидетельствует о том, что и кольцевая, и ячеистая схемы подходят для построения сетей обмена в мультипроцессорных системах. Кольцевые системы характеризуются более простой реализацией, но худшей масштабируемостью. Поэтому они более пригодны для систем с относительно небольшим количеством процессоров (до нескольких сотен). Ячеистые сети подходят и для больших, и для малых систем. А для совсем маленьких сетей, содержащих до 16 процессоров, самым эффективным решением является архитектура с общей шиной или координатной коммутацией.

Читателя может заинтересовать, каков масштаб систем, имеющих реальное практическое применение. Большинство мультипроцессорных систем относительно невелики. Многие из них содержат от 4 до 128 процессоров. Однако существуют и очень большие системы, включающие тысячи процессоров. Впрочем, сфера их применения ограниченна.

12.4.9. Сети смешанной топологии

Мы рассмотрели несколько сетевых топологий и попытались описать достоинства и недостатки каждой из них. Разработчики мультипроцессорных систем стараются добиться максимальной производительности за приемлемую стоимость. Поэтому во многих коммерческих системах используются смешанные топологии — их разработчики постарались соединить достоинства нескольких топологий в единой системе. Например, отличным способом объединения нескольких процессоров является комбинация шинной топологии и топологии с координатной коммутацией. Часто встречаются процессорные кластеры, обычно содержащие от 2 до 8 процессоров, соединенные шиной или координатным коммутатором. Такие кластеры выступают в роли узлов сети, объединенных в большую

систему с помощью подходящей топологии. В системе AV25000 производства Data General используются кластерные узлы, процессоры которых связаны общей шиной. В единую сеть эти узлы соединяются по кольцевой схеме. В системе Exemplar V2600 производства Hewlett-Packard узлы тоже соединены при помощи кольцевой сети, а процессоры в каждом из них объединены между собой посредством координатного коммутатора. В системе AlphaServer SC производства Compaq узлы сети соединяются в толстое дерево, а процессоры внутри узлов соединены путем координатной коммутации.

12.4.10. Симметричные мультипроцессорные системы

Рассмотрим мультипроцессорную систему, в которой все процессоры имеют одинаковые возможности доступа к модулям памяти и устройствам ввода-вывода, так что с точки зрения операционной системы процессоры являются абсолютно идентичными. Если любой из процессоров может выполнять и ядро операционной системы, и пользовательские программы, система называется *симметричной мультипроцессорной системой* (Symmetric Multiprocessor, SMP). В такой системе любой процессор может обрабатывать внешние прерывания и инициировать операцию ввода-вывода для любого устройства ввода-вывода.

Симметричные мультипроцессорные системы обычно реализуются на основе шинной сети или сети с координатной коммутацией. Они часто используются в качестве узлов более крупных мультипроцессорных систем. Например, узлы SMP входят в состав описанных выше мультипроцессоров Exemplar V2600 и AlphaServer SC.

12.5. Организация памяти в мультипроцессорных системах

В главе 5 было показано, какое большое влияние на производительность оказывает организация памяти в однопроцессорной системе. Сказанное верно и для мультипроцессорных систем. Для того чтобы использовать свойство локализации ссылок, в каждый процессор обычно включают первичный и вторичный кэши. Если система построена так, как на рис. 12.2, каждый процессорный модуль может быть соединен с коммуникационной сетью способом, показанным на рис. 12.12. На этом рисунке изображен только вторичный кэш, поскольку мы считаем, что первичный кэш интегрирован в микросхему процессора. При обращении к модулям памяти используется *глобальное адресное пространство*, в котором каждому модулю назначен свой диапазон физических адресов. В такой системе *с общей памятью* процессоры имеют равный доступ ко всем модулям памяти. С точки зрения программного обеспечения это простейший способ использования адресного пространства.

В мультипроцессорных системах типа NUMA (рис. 12.3) каждый узел содержит процессор и модуль памяти. Наиболее естественный способ реализации такого узла показан на рис. 12.13. Для подобных систем удобно использовать глобальное адресное пространство. И снова каждый процессор может обратиться

к любому модулю памяти, хотя на доступ к локальным модулям памяти глобального адресного пространства уходит меньше времени, чем на доступ к удаленным модулям.

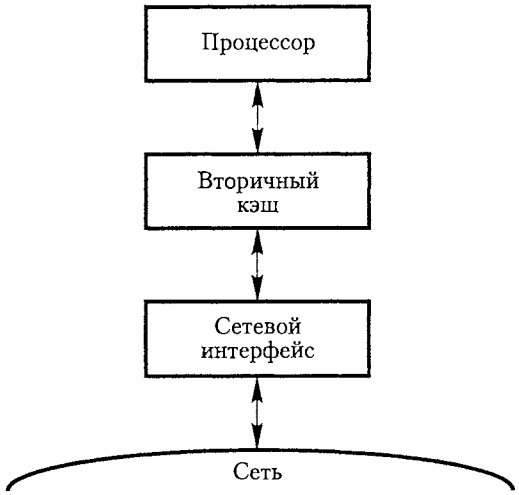


Рис. 12.12. Процессорный узел в мультипроцессорной системе, показанной на рис. 12.2

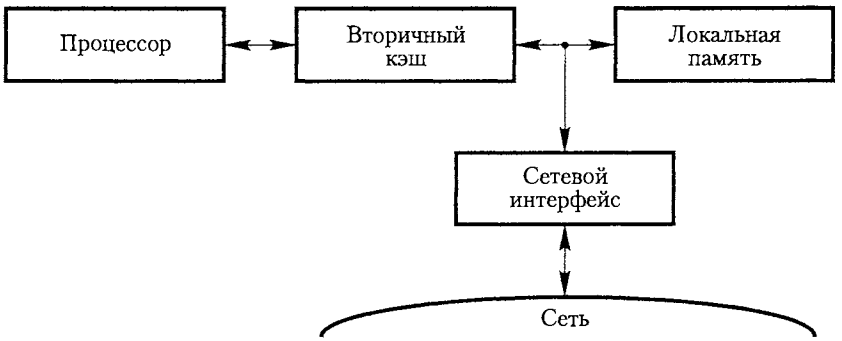


Рис. 12.13. Структура узла мультипроцессора, изображенного на рис. 12.3

В системе с распределенной памятью (рис. 12.4) процессоры могут обращаться напрямую только к своей локальной памяти. Поэтому каждый модуль памяти составляет *локальное адресное пространство* одного процессора; глобального адресного пространства нет вообще. Любое взаимодействие между программами или процессами, выполняющимися на разных процессорах, осуществляется при помощи *сообщений*, передаваемых между процессорами. При таком виде взаимодействия каждый процессор рассматривает коммуникационную сеть как устройство ввода-вывода. В таком случае каждый узел системы можно рассматривать как отдельный компьютер. Поэтому системы подобного типа также называются *мультикомпьютерными*. Данная архитектура представляет собой простейший способ соединения множества компьютеров в большую систему. Взаимодействие между

задачами, выполняющимися на разных компьютерах, осуществляется довольно медленно, поскольку для обмена сообщениями требуется программное обеспечение. Этот тип систем рассматривается в разделе 12.7.

Когда одни и те же данные используются множеством процессоров, необходимо гарантировать, что все процессоры будут работать с одними и теми же значениями элементов данных. В связи с этим наличие множества кэшей в системе с общей памятью составляет серьезную проблему. В разных кэшах могут находиться разные копии одних и тех же данных. Когда процессор изменяет элемент данных в собственной кэш-памяти, это изменение должно быть внесено во все остальные кэши, в которых имеются копии этих данных. Или же все остальные копии должны быть помечены как недостоверные. Иными словами, необходимо, чтобы общие данные были согласованы (когерентны) во всех кэшах системы. Самое распространенное решение этой проблемы мы рассмотрим в разделе 12.6.2.

12.6. Программный параллелизм и общие переменные

Во введении к этой главе говорилось, что сложную задачу не всегда легко разбить на параллельно выполняемые подзадачи. Однако есть особые случаи, когда это сделать достаточно просто. В частности, если главная задача представляет собой набор независимых программ, эти программы легко выполнить на разных процессорах. И если программы не блокируют друг друга, соревнуясь за устройства ввода-вывода, мультипроцессор оказывается полностью загруженным.

Кроме того, программу легко разделить на параллельно выполняемые задачи при использовании языка высокого уровня, позволяющего программисту явно выделить отдельные задачи. Такая конструкция часто называется PAR-сегментом. Она показана на рис. 12.14. Между управляющими командами PARBEGIN и PAREND находится список процедур, от Proc1 до Proc*K*, которые могут выполняться параллельно. Рассмотрим порядок реализации этой программы. После завершения сегмента программы, предшествующего команде PARBEGIN, может немедленно начаться выполнение любой из *K* параллельных процедур или же всех этих процедур, что зависит от количества свободных процессоров. Последовательность их запуска может быть любой. Выполнение части программы, следующей за командой PAREND, разрешается только после завершения выполнения всех или *K* параллельных процедур.

```
⋮
PARBEGIN  ┐
Proc1;    │
Proc2;    │
⋮         ├─ PAR-сегмент
ProcK;    │
PAREND    ┘
⋮
```

Рис. 12.14. Программная конструкция для параллельного выполнения

Если мультипроцессорная система выполняет только эту программу, за эффективное использование процессоров отвечает программист. Степенью параллелизма PAR-сегментов, *K*, и отношением их общего размера к размеру последовательных сегментов определяется эффективность работы мультипроцессорной системы.

Наиболее эффективное использование мультипроцессорных систем достигается путем применения компиляторов, способных автоматически определять фрагменты пользовательских программ, которые можно запускать параллельно. Программист обычно представляет программу как набор последовательно производимых операций. Но, несмотря на это, существует много возможностей параллельной реализации различных групп команд. Простейшим примером является многократно выполняемый программный цикл. Если данные, используемые на разных итерациях, независимы, итерации можно осуществлять параллельно. Если же при первом прохождении по циклу генерируются данные для второго прохождения и т. д., параллельная работа не возможна. Компилятор должен выявлять зависимости по данным и решать, какие операции можно выполнить параллельно, а какие — нельзя. Разработка компиляторов, разбивающих программу на параллельно выполняемые фрагменты, — задача не из простых. И даже после того, как такие фрагменты будут выделены, их еще нужно оптимально распределить между процессорами, так как свободных процессоров может быть меньше, чем фрагментов. Мы не будем обсуждать вопрос распределения подзадач между процессорами и планирования последовательности их реализации, а лишь поговорим о доступе к общим переменным, которые модифицируются параллельно выполняемыми подзадачами в мультипроцессорной системе.

12.6.1. Доступ к общим переменным

Предположим, что мы определили две задачи, которые могут параллельно выполняться мультипроцессорной системой. Эти задачи почти полностью независимы, но время от времени они считывают и модифицируют некоторую общую переменную, хранящуюся в глобальной памяти. Пусть, например, общая переменная SUM представляет баланс некоторого счета. Этот счет должен обновляться несколькими задачами, выполняющимися разными процессорами. Каждая задача производит с переменной SUM следующие действия: считывает ее текущее значение, выполняет некоторую операцию с его использованием, а результат сохраняет в переменной SUM. Нетрудно представить себе, какие ошибки могут возникнуть, если такие операции с переменой SUM будут осуществляться задачами T1 и T2, выполняющимися в параллельном режиме процессорами P1 и P2. Предположим, что обе задачи, T1 и T2, считывают текущее значение переменной SUM, скажем 17, и каждая по отдельности его модифицирует. Задача T1 прибавляет к нему 5, получая 22, а задача T2 вычитает из него 7, результатом чего является число 10. Затем каждая из задач заносит в переменную SUM свой результат — сначала это действие выполняет задача T2, затем задача T1. Теперь в переменной SUM хранится число 22, что является ошибкой, так как на самом деле в этой переменной должно находиться значение 15 (17 + 5 − 7), которое получится при строго последовательном проведении изменений.

Для того чтобы обеспечить правильную работу с переменной SUM, каждая задача должна получать к ней монопольный доступ на все время выполнения

последовательности операций чтения, модификации и записи. С этой целью может быть задействована глобальная *переменная блокировки* LOCK и машинная команда Test and Set. Переменная LOCK может принимать значение 0 или 1. Мы используем ее для того, чтобы гарантировать, что никакие две задачи не смогут одновременно получить доступ к переменной SUM. Последовательность команд, для выполнения которой требуется монопольный доступ к некоторой переменной, называется *критической секцией* программы. Переменная LOCK используется следующим образом. Если ни одна из задач не находится в состоянии выполнения критической секции, оперирующей переменной SUM, переменная LOCK равна 0. Когда какая-либо из задач собирается модифицировать переменную SUM, она сначала проверяет значение переменной LOCK, а затем устанавливает таковое в 1 независимо от ее исходного значения переменной. Если исходным значением является 0, задача приступает к работе с переменной SUM, поскольку это означает, что никакая другая задача с этой переменной в данный момент не работает. Если же исходное значение переменной LOCK равняется 1, значит, с переменной SUM работает другая задача. В таком случае первая задача должна находиться в состоянии ожидания до тех пор, пока вторая не установит значение переменной LOCK в 0, и лишь затем она может приступать к собственной операции с этой переменной. Все перечисленные операции с переменной LOCK выполняются командой Test and Set. Эта команда в соответствии со своим названием незаметно для программы выполняет проверку и установку переменной LOCK. В процессе ее работы соответствующий модуль памяти не должен отвечать на запросы доступа от других процессоров.

```
                    Task T1
                      ⋮
LOCK1   TAS.B   LOCKBYTE
        BMI     LOCK1
        «Модификация SUM»   } Критическая секция
        CRL.B   LOCKBYTE
                      ⋮

                    Task T2
                      ⋮
LOCK2   TAS.B   LOCKBYTE
        BMI     LOCK2
        «Модификация SUM»   } Критическая секция
        CRL.B   LOCKBYTE
                      ⋮
```

Рис. 12.15. Совместный доступ к критической секции программы

Рассмотрим конкретный пример — действие команды Test and Set (или TAS) микропроцессора 68000 компании Motorola. Один из операндов этой команды имеет размер 1 байт. Предположим, что данный операнд хранится в памяти по адресу LOCKBYTE. Бит b_7 — старший бит этого операнда — выполняет роль описанной выше переменной LOCK. При осуществлении командой TAS операций проверки и установки бита b_7 прерываний не происходит. Флагу кода условия N (отрицательное значение) присваивается исходное значение бита b_7. Если флаг N равен 0, по завершении выполнения команды TAS программа может войти в критическую секцию, а если он равен 1, программа должна подождать, пока он не примет значение 0. На рис. 12.15 показано, как две задачи, T1 и T2, манипулируют переменной LOCKBYTE для входа в критическую секцию кода, где они обновляют общую переменную SUM. За командой TAS следует команда условного перехода. При N = 1 опять производится переход к команде TAS, в результате чего получается цикл ожидания, в котором эта команда выполняется над операндом по адресу LOCKBYTE до тех пор, пока бит b_7 не примет значение 0. Если b_7 = 0, переход к команде TAS не осуществляется и программа входит в критическую секцию. По завершении выполнения критической секции переменная LOCKBYTE очищается. В результате бит b_7 устанавливается в 0 и ожидающая этого программа может войти в свою критическую секцию.

Команда TAS является примером простой машинной команды, предназначенной для реализации блокировки. Подобные команды имеются в большинстве компьютеров. Они могут выполнять и дополнительные функции, такие как условные переходы на основе результатов проверки.

12.6.2. Согласованность кэша

Использование общих данных в мультипроцессорной системе приводит к еще одной серьезной проблеме, связанной с наличием в нескольких кэшах копий одних и тех же данных. Когда процессор записывает общую переменную в ее собственный кэш, во всех остальных кэшах, содержащих ее копии, оказываются устаревшие и неверные данные. Эти кэши должны быть проинформированы об изменениях, чтобы они могли либо обновить свои копии, либо пометить их как недостоверные. Состояние, когда все кэшируемые копии общих данных имеют одинаковые значения, называется *согласованностью* (*когерентностью*) *кэша*.

В главе 5 рассматривались два основных подхода к выполнению операций записи в кэш. При сквозной записи данные одновременно изменяются и в кэше, и в основной памяти. При обратной записи данные изменяются только к кэше, а копия в основной памяти обновляется при замене блока данных в кэше. Аналогичные технологии могут использоваться и в мультипроцессорных системах.

Протокол сквозной записи

Протокол сквозной записи может быть реализован двумя способами. В соответствии с первым из них значения в других кэшах обновляются, а в соответствии со вторым — помечаются как недостоверные.

Для начала рассмотрим протокол *сквозной записи с обновлением*. Когда процессор записывает в кэш новое значение, такое же значение помещается и в модуль

памяти, содержащий измененный блок кэша. Поскольку копии этого блока могут присутствовать и в других кэшах, они также обновляются. Простейшим способом выполнения таких изменений является широковещательная передача записанных данных всем процессорным модулям системы. Каждый процессорный модуль получает обновленные данные, и если в его первичном или вторичном кэше имеется соответствующий блок, обновляет таковой.

Во втором варианте протокола сквозной записи копии помечаются как *недостоверные*. Когда процессор записывает в свой кэш новое значение, такое же значение записывается в модуль памяти, а затем все остальные его копии в других кэшах помечаются как недостоверные. Сообщения о недостоверности данных рассылаются по системе путем широковещательной передачи.

Протокол обратной записи

В протоколе обратной записи несколько процессоров могут прочитать из памяти в свои кэши копии одного и того же блока данных. Если какой-либо из процессоров захочет изменить свой блок, сначала он должен стать его монопольным владельцем. Монопольный доступ к блоку предоставляется процессору тем модулем памяти, в котором этот блок расположен, и после этого все остальные копии такового, включая копию в основной памяти, помечаются как недостоверные. Теперь владелец блока может свободно изменять его содержимое. Когда другой процессор захочет прочитать этот блок, данные будут ему пересланы от текущего владельца. По окончании процесса изменения данные будут возвращены в исходный модуль памяти, который снова получит права владельца соответствующего блока.

Обратная запись вызывает меньший трафик, чем сквозная, поскольку прежде чем блок кэша потребуется другим процессорам, получивший его процессор может несколько раз записать в него данные.

До сих пор предполагалось, что запросы на обновление и недостоверные данные распространяются по сети путем широковещательной передачи. Но на практике это зависит от архитектуры конкретной сети. Для широковещательной передачи лучше всего подходит шинная архитектура, описанная в разделе 12.4.1. В маленьких мультипроцессорах с общей шиной для обеспечения согласованности кэша может использоваться технология, называемая отслеживанием.

Кэш с отслеживанием

В системе с общей шиной все транзакции между процессором и модулями памяти выполняются через шину. В результате они передаются всем соединенным с шиной устройствам. Предположим, что в кэше каждого процессора имеется управляющая схема, наблюдающая за транзакциями на шине, в которых участвуют другие процессоры. Для работы с кэш-памятью используется протокол обратной записи.

Когда процессор в первый раз выполняет запись блока в свой кэш, это блок помечается как измененный (dirty) и информация о записи распространяется по сети. Модуль памяти и все остальные кэши помечают свои копии измененного блока как недостоверные. Выполнивший запись процессор с этого момента является владельцем блока. Он может вносить в него изменения без их распространения по сети. Если другой процесс выдаст запрос на чтение данного блока, модуль

памяти окажется не в состоянии на него ответить, поскольку в нем нет достоверной копии. Однако текущий владелец блока тоже увидит этот запрос на шине и должен будет предоставить выдавшему его процессору достоверные данные. Новые данные, содержащиеся в отправленном владельцем широковещательном сообщении, считываются модулем памяти, после чего последний обновляет свой блок. Затем владелец блока помечает свою копию как чистую (неизмененную). Теперь во всех кэшах и в памяти находятся обновленные данные, и весь процесс при необходимости повторяется. Если измененный блок в кэше одного из процессоров должен быть заменен новым, выполняется операция обратной записи в модуль памяти.

Если два процессора хотят одновременно записать данные в один и тот же блок, доступ к шине предоставляется только одному из них (он становится владельцем блока). В результате копия этого блока в другом процессоре помечается как недостоверная. Позднее второй процессор может повторить свой запрос. Такая последовательная обработка запросов на запись гарантирует корректное изменение данных двумя процессорами в одном и том же блоке.

Описанная схема основана на способности контроллеров кэша следить за активностью шины и выполнять соответствующие действия. Поэтому кэш-память подобного типа называется *кэшем с отслеживанием* (snoopy-cache).

С точки зрения производительности важно, чтобы функция отслеживания не мешала нормальному функционированию процессора и его кэш-памяти. Проблема состоит в том, что при каждом появляющемся на шине запросе контроллер кэша должен обращаться к хранящимся в нем тегам, чтобы узнать, присутствует ли там искомый блок. В большинстве случаев ответ оказывается отрицательным. Поэтому, с тем чтобы изолировать процесс отслеживания от выполнения остальных функций узла, кэш снабжается двумя наборами тегов, содержащими одну и ту же информацию о блоках. Схемы отслеживания могут обращаться к своему набору тегов независимо от остальных схем управления кэш-памятью.

Хотя концепция кэша с отслеживанием эффективна и проста в реализации, она подходит только для систем с общей шиной. В более крупных мультипроцессорах приходится использовать более сложные технологии согласования кэш-памяти.

Схемы на основе каталогов

Схема согласования кэша при помощи широковещательной передачи сообщений об обновленных или недостоверных данных не годится для больших систем. Широковещательные сообщения вызывают слишком большой трафик, несмотря на то что копии каждого конкретного блока обычно присутствуют всего в нескольких кэшах. Альтернативой может служить создание *каталога* с информацией о кэшах, хранящих копии каждого из блоков. Для реализации такого каталога можно включить в каждый блок дополнительные биты, указывающие, в каких кэшах имеются его копии. Тогда вместо широковещательной рассылки информации об изменениях модуль памяти может направлять процессорным модулям отдельные или групповые сообщения. Конечно, дополнительные биты в модулях памяти увеличивают их стоимость. Предлагались различные схемы реализации каталога, и некоторые из них были воплощены в существующих мультипроцессорных системах.

Стандарт SCI

Один из способов обеспечения согласованности кэш-памяти был стандартизирован Институтом инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers, IEEE). Он приведен в стандарте SCI (Scalable Coherent Interface — масштабируемый когерентный интерфейс), который определяет структуру мультипроцессора, обеспечивающую быстрое распространение сигналов, масштабируемую архитектуру, согласованность кэша и простую реализацию. В сети обмена используются соединения «точка-точка», а коммуникационный протокол основывается на принципе «один источник запроса — один приемник». Каждый пакет высылается одним узлом-источником и адресуется единственному узлу-приемнику. Если пакет принят адресатом, последний возвращает пакет с положительным подтверждением. Если пакет не принят, высылается отрицательное подтверждение, вызывающее повторную отправку пакета.

Согласованность кэш-памяти достигается при помощи протокола на основе распределенного каталога. Для каждого блока кэша, содержащего общие данные, создается двусвязный список. Каждый процессорный узел, кэширующий этот блок, помещает в список указатели на использующие блок предыдущий и следующий узлы. Эти указатели записываются в тег блока кэш-памяти. Концом созданного таким образом двусвязного списка является модуль памяти, где хранится данный блок. Когда очередной узел обращается к модулю памяти, чтобы прочитать нужный блок, этот узел становится новым началом списка, а каталог предыдущего начала списка обновляется — указатель на предыдущий узел заменяется указателем нового узла. Доступ к памяти для записи разрешается только первому узлу списка (точнее, его началу). Если другой узел захочет выполнить запись, он должен будет записать себя в начало списка и очистить остальные элементы списка.

SCI-схема согласования кэша хорошо масштабируется, так как с увеличением списка узлов требования к объему памяти для его хранения ни в основной памяти, ни в кэш-памяти процессоров не увеличиваются. А недостатком этой схемы является необходимость хранения дополнительной информации для каждого блока.

Хотя стандарт SCI не определяет конкретную топологию сети обмена, наиболее подходящей для него является кольцевая топология. Она реализована в мультипроцессорах Exemplar V2600 производства Hewlett-Packard и AV25000 производства Data General вместе с описанным в этом разделе протоколом согласования кэша.

Мультипроцессоры cc-NUMA

Проблема согласованности кэша является одной из важнейших проблем при разработке мультипроцессорных систем и предметом самых интенсивных исследований. Мы коротко описали несколько основных схем согласования кэша, но более подробное обсуждение этого вопроса выходит за рамки нашей книги.

Согласованность кэша может обеспечиваться как аппаратными, так и программными средствами. Но более высокая производительность достигается при использовании аппаратных средств. Поэтому в большинстве современных мультипроцессоров NUMA согласованность кэша реализуется аппаратно. Такие системы называются системами *NUMA с согласованной кэш-памятью* (cache coherent NUMA, cc-NUMA).

12.6.3. Блокировка и согласованность кэш-памяти

Следует отметить, что схемы управления блокировкой для доступа к переменным и схемы согласования кэш-памяти независимы и играют одинаково важную роль. Рассмотрим ситуацию, когда согласованность кэша поддерживается посредством сквозной записи и при этом запись общих переменных сопровождается обновлением кэша. Предположим, что содержимое переменной SUM в примере из раздела 12.6.1 считывается в кэши двух процессоров, выполняющих задачи T1 и T2. Если команды чтения являются частью последовательности операций обновления переменной и для их взаимоисключения не используется механизм блокировок, описанный в разделе 12.6.1, тогда опять может произойти обсуждавшаяся в указанном разделе ошибка. Если, как и раньше, задача T1 запишет новое значение ошибки, в переменной SUM окажется неправильное значение 22. В этом случае достигается согласованность кэш-памяти, но для получения правильных результатов все же необходима блокировка.

12.7. Мультикомпьютерные системы

В разделе 12.5 было введено понятие мультикомпьютерных систем. Теперь мы подробно рассмотрим основные особенности таких систем.

Мультикомпьютерная система структурирована так, как показано на рис. 12.4. Каждый процессорный узел в этой системе представляет собой самодостаточный компьютер, взаимодействующий с другими процессорными узлами при помощи передаваемых по сети сообщений. Системы подобного типа часто называют системами *с передачей сообщений*, в противоположность системам с общей памятью, рассматривавшимся в первой части данной главы.

В мультикомпьютерных системах требования к коммуникационной сети менее строги, чем в мультипроцессорных системах с общей памятью. Сеть системы с общей памятью должна обладать достаточно высоким быстродействием, иметь широкую полосу пропускания, поскольку процессорные модули часто обращаются к удаленным модулям памяти, из которых состоит общая память. Медленная сеть заметно снизит производительность такой системы.

Сообщения в мультикомпьютерной системе передаются гораздо реже, поэтому в ней нет такого интенсивного трафика. Значит, для связи процессорных узлов может использоваться более простая и дешевая сеть. Появление терминов *сильносвязанная система* и *слабосвязанная система*, используемых соответственно по отношению к системам с общей памятью и системам с передачей сообщений, обусловлено именно разницей в интенсивности взаимодействия узлов сети.

В мультикомпьютерных системах может использоваться любая из сетей, описанных в разделе 12.4. Так как трафик в сети относительно невелик, ее физическая реализация обычно получается недорогой. Узлы сети часто соединяются линиями с последовательной передачей битов, управляемыми интерфейсами ввода-вывода. Интерфейсная схема компьютера-источника считывает из памяти сообщение посредством механизма прямого доступа к памяти, преобразует его в последовательный формат и передает по сети компьютеру-приемнику. Адреса источника и места назначения включаются в заголовок сообщения для использования

в процессе маршрутизации. Сообщение достигает компьютера-приемника и записывается в буфер памяти при помощи его интерфейса ввода-вывода.

В 1980-х годах были очень популярны сети гиперкубической топологии. Сети подобного типа использовались в нескольких мультипроцессорных системах с передачей сообщений, и в них применялась последовательная передача битов. Примерами таких систем являются iPSC компании Intel, NCUBE/ten компании NCUBE и CM-2 компании Thinking Machines. Позднее, в начале 1990-х годов, для систем с передачей сообщений, равно как для систем с общей памятью, стали использоваться другие топологии. Например, в системе CM-2 производства Thinking Machines применялась сеть с передачей сообщений топологии толстого дерева, ширина соединений в которой равнялась 4. В системе Paragon компании Intel использовалась ячеистая сеть с шириной соединений 16. Для более эффективной передачи сообщений в каждый узел сети помещалось специальное коммуникационное устройство. Например, в системе Paragon использовался процессор сообщений, освобождавший процессор приложений от участия в передаче данных.

12.7.1. Локальные сети

Поскольку коммуникационные требования мультикомпьютерной системы относительно невелики, специализированную сеть обмена можно заменить какой-нибудь стандартной сетью, предназначенной для более универсальных коммуникационных нужд. Для соединения компьютерного оборудования разработано множество сетей. Сети, узлы которых расположены на небольшом расстоянии друг от друга (в пределах нескольких километров), называются *локальными вычислительными сетями*, ЛВС (Local Area Network, LAN). Сети большего размера, распространяющиеся на тысячи километров, известны как *сети с протяженными линиями связи* (Long Haul Network, LHN) или *глобальные вычислительные сети* (Wide Area Network, WAN).

В самых популярных локальных сетях используется шинная или кольцевая топология. В качестве передающей среды в них могут быть задействованы витая пара, коаксиальный кабель или волоконно-оптический кабель. Между узлами выполняется последовательная передача битов, скорость которой варьируется от десятков до сотен мегабит в секунду. По одному соединению не может одновременно передаваться более одного пакета. Полям данных в пакете предшествуют адреса источника и места назначения, а начало и конец пакета отмечаются с помощью специальных разделителей. В общем случае пакеты имеют переменную длину от десятков до тысяч байтов.

Для обеспечения передачи пакетов между любыми двумя узлами сети требуется протокол управления распределенным доступом. Мы коротко рассмотрим основные идеи двух популярных протоколов — Ethernet и Token Ring. Эти протоколы подробно описаны в стандартах IEEE.

12.7.2. Протокол Ethernet

Шинный протокол *Ethernet*, называемый также *протоколом с множественным доступом с контролем несущей и обнаружением конфликтов* (Carrier Sense Multiple Access with Collision Detection, CSMA/CD) с концептуальной точки зрения

является одним из простейших сетевых протоколов. Рассмотрим его работу. Прежде чем начать передачу сообщения, подключенное к шине устройство ждет ее освобождения. В течение 2τ (где τ — это время распространения сигнала между двумя концами шины) устройство наблюдает за процессом передачи своего сообщения по шине. Если за этот промежуток времени никакого искажения передаваемого им сигнала не происходит, устройство продолжает передачу сообщения. Если же обнаружится изменение сигнала, вызванное началом передачи от некоторого другого устройства, оба устройства прекращают передачу. Взаимоискажающее наложение двух передаваемых по шине сигналов называется *конфликтом* (*столкновением* или *коллизией*), а временной интервал 2τ — окном конфликта.

Сообщения, уничтоженные в результате конфликта, должны быть переданы повторно. Если участвующие в конфликте устройства попытаются сделать это немедленно, их пакеты почти наверняка столкнутся снова. Поэтому каждое из устройств на некоторое время (интервал времени выбирается случайно) прекращает свою работу, а потом, дождавшись момента освобождения шины, начинает повторную передачу. Если случайные интервалы в несколько раз больше промежутка времени 2τ, вероятность повторного конфликта минимальна.

12.7.3. Протокол Token Ring

Протокол Token Ring («маркерное кольцо») применяется для кольцевых сетей. При его использовании по сети постоянно циркулирует одно короткое закодированное сообщение, называемое *маркером*. Прибытие этого маркера на узел сети означает разрешение на передачу данных. Если узлу нечего передавать, он пересылает маркер следующему узлу с как можно меньшей задержкой. Если же узел готов к передаче, он блокирует маркер и вместо него отправляет пакет информации, предваряемый закодированным флагом заголовка. Пакет передается по кольцу от узла к узлу. По достижении пакетом узла назначения его содержимое считывается и копируется во внутренний буфер узла-адресата. Далее пакет продолжает путешествие по кольцу, пока не вернется на исходный узел. Здесь он уничтожается, а исходный узел, закончив передачу пакета, освобождает маркер и отправляет его дальше по кольцу. Пакеты в такой сети имеют переменную длину, ограниченную только объемом памяти буфера на каждом из узлов.

Мы рассматриваем стандартные локальные сети в контексте мультикомпьютерных систем не потому, что они могут использоваться в готовых системах, которые обсуждались до сих пор, а потому, что мультикомпьютерная система может быть создана на основе рабочих станций, объединенных при помощи локальной сети.

12.7.4. Сеть рабочих станций

В настоящее время в большинстве коммерческих, образовательных и правительственных организаций компьютерные системы состоят из множества рабочих станций. Эти рабочие станции обычно объединяются в локальные сети, обеспечивающие доступ к файловым серверам, принтерам и специализированным компьютерным ресурсам (рис. 12.16).

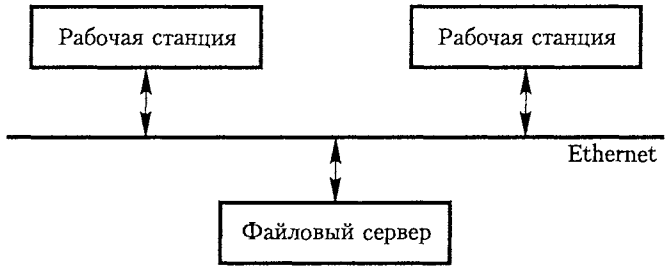


Рис. 12.16. Типичная сеть, соединяющая рабочие станции

Хотя каждая рабочая станция используется как отдельный компьютер, их набор может рассматриваться как мультикомпьютерная система. Все, что для этого нужно, — это программное обеспечение, поддерживающее параллельную обработку. Конечно, между такой системой и настоящей мультипроцессорной системой с передачей сообщений существуют значительные различия. В частности, взаимодействие через локальную сеть осуществляется медленнее, главным образом потому, что в обмен сообщениями между программами, работающими на разных компьютерах, должна вмешиваться операционная система. Это означает, что сеть рабочих станций функционирует не настолько эффективно, как единая система со специализированной сетью обмена. Но зато такая сеть доступна и проста в реализации.

12.8. Общая память и передача сообщений

В предыдущих разделах были описаны аппаратные аспекты мультипроцессорных систем с общей памятью и с передачей сообщений. Теперь мы кратко рассмотрим два этих типа систем с точки зрения программиста, который разрабатывает приложение, поддерживающее параллельную обработку. Далее будет приведен небольшой пример с использованием двух процессоров. Это позволит упростить обсуждение и четче выделить основные идеи.

Предположим, необходимо вычислить скалярное произведение двух N-элементных векторов. Последовательная программа решения этой задачи предназначена для выполнения на одном процессоре (рис. 12.17). Логика программы понятна. Инструкции считывания загружают с диска (или с какого-либо другого устройства ввода-вывода) в основную память значения двух векторов. Эта задача выполняется операционной системой. Давайте попробуем распараллелить задачу для реализации на двух процессорах. Сделать это можно в цикле, который вычисляет произведение очередной пары элементов, а результаты суммирует.

12.8.1. Система с общей памятью

Первый вариант программы для двух процессоров показан на рис. 12.18. В случае запуска программы на одном процессоре она загружает векторы в память и присваивает переменной *dot_product* значение 0. При параллельном выполнении программы на двух процессорах часть вычислений, требуемых для получения

скалярного произведения, необходимо возложить на один из них. Для этого мы создаем отдельный поток, предназначенный для выполнения на таком процессоре.

```
integer array a[1...N], b[1...N]

integer dot_product
...
read a[1...N] from vector_a
read b[1...N] from vector_b
dot_product :=0
do_dot(a, b)
print dot_product
...
do_dot (integer array x[1...n], integer array y[1...N])
  for k:= 1 to N
    dot_product := dot_product + x[k]*y[k]
  end
end
```

Рис. 12.17. Последовательная программа вычисления скалярного произведения

```
shared integer array a[1...N], b[1...N]

shared integer dot_product
shared lock dot_product_lock
shared barrier done
...
read a[1...N] from vector_a
read b[1...N] from vector_b
dot_product :=0
create_thread (do_dot, a, b)
do_dot (a, b)
print dot_product
...
do_dot (integer array x[1...N], integer array y[1...N])
  private integer id
  id := mypid()
  for k :=(id*N/2) + 1 to (id+1)*N/2
    lock (dot_product_lock)
      dot_product := dot_product + x[k] * y[k]
    unlock (dot_product_lock)
  end
  barrier (done)
end
```

Рис. 12.18. Первый вариант программы вычисления скалярного произведения на двух процессорах в системе с общей памятью

Поток — это независимый фрагмент программы. Потоки могут соответствовать разным фрагментам кода программы или одному и тому же фрагменту, выполняемому несколько раз в различных условиях. Главное, что они могут выполняться параллельно, как отдельные программы, поэтому их можно запускать на разных процессорах. Вместе с тем потоки являются частями одной и той же программы, реализуемыми в одном и том же адресном пространстве. В типичном однопроцессорном окружении каждая программа имеет один поток выполнения.

В программе на рис. 12.18 новый поток создается посредством инструкции create_ thread. Вызвав процедуру do_dot, этот поток завершает свою работу. Операционная система присваивает новому потоку идентификационный номер 1. Далее первый процессор выполняет инструкцию do_dot (a,b) как поток 0. Инструкция id := mypid() присваивает переменной id идентификационный номер потока. С помощью переменной id в цикле for мы определяем, какая половина векторов a и b должна обрабатываться данным потоком.

Критической секцией процедуры do_dot является код, изменяющий значение переменной dot_product. Каждый поток должен получать монопольный доступ к указанной переменной. Для этого используется описанный в разделе 12.6.1 механизм блокировок. Поток 0 не идет далее инструкции-барьера barrier в процедуре do_dot, пока другой поток не достигнет той же синхронизационной точки. Это необходимо для того, чтобы оба потока завершили свои вычисления до того, как поток 0 сможет напечатать конечный результат. Инструкцию-барьер можно реализовать двумя способами. Простейший подход заключается в использовании общей переменной, такой как done (рис. 12.18). Она инициализируется значением количества потоков (в нашем примере их два), и когда каждый поток достигает барьера, ее значение уменьшается на единицу.

У программы на рис. 12.18 имеется один существенный недостаток. Используемый в ней механизм блокировок не позволяет по-настоящему параллельно выполнять два потока, поскольку оба потока постоянно пытаются записать данные в одну и ту же общую переменную dot_product, а делать это одновременно они не могут. Таким образом, потенциально параллельные вычисления на самом деле выполняются последовательно.

Чтобы добиться настоящего параллелизма, можно так модифицировать программу, как показано на рис. 12.19. Вместо использования в цикле for общей переменной dot_product мы задействовали локальную переменную local_dot_ product, в которой накапливается частичное скалярное произведение, вычисляемое каждым из потоков. Вход в критическую секцию, где каждый поток обновляет общую переменную dot_product, производится только по окончании цикла. После такой модификации циклы for обоих потоков действительно могут выполняться параллельно.

Приведенный пример легко распространить на большее количество процессоров. С этой целью достаточно создать больше потоков. В цикле for на основе значения переменной id будет определяться диапазон элементов, используемых для вычислений каждым потоком.

Эффективность работы программы на рис. 12.19 зависит от размера векторов данных. Чем они больше, тем более эффективен описанный подход. Для малых же векторов затраты на создание дополнительных потоков и их синхронизацию перевешивают преимущества параллельного выполнения.

```
shared integer array a[1...N], b[1...N]

shared integer dot_product
shared lock dot_product_lock
shared barrier done
...
read a[1...N] from vector_a
read b[1...N] from vector_b
dot_product :=0
create_thread (do_dot, a, b)
do_dot (a, b)
print dot_product
...
do_dot (integer array x[1...N], integer array y[1...N])
  private integer local_dot_product
  private integer id
  id := mypid()
  local_dot_product := 0
  for k :=(id*N/2) + 1 to (id+1)*N/2
      local_dot_product := local_dot_product + x[k] * y[k]
  end
  lock (dot_product_lock)
    dot_product := dot_product + local_dot_product
  unlock (dot_product_lock)
  barrier (done)
end
```

Рис. 12.19. Эффективная программа для вычисления скалярного произведения на двух процессорах в системе с общей памятью

12.8.2. Система с передачей сообщений

В случае распределенной памяти каждый процессор обращается к собственной памяти. Если выполнять нашу программу на двух процессорах в такой системе, векторы должны быть явно разделены на две части, каждая из которых будет храниться в памяти одного процессора. Каждая копия программы будет иметь доступ только к своим данным. Архитектура для выполнения таких приложений называется архитектурой *с одной программой и несколькими потоками данных* (Single Program, Multiple Data, SPMD). Читатель должен понимать различие между системой SPMD и описанной в разделе 12.1.1 системой SIMD, где все процессоры в конкретный момент времени выполняют одну и ту же команду.

Пример программы типа SPMD приведен на рис. 12.20. Вектор данных должен быть сначала загружен в локальные модули памяти двух процессоров. Программа, которой присвоен идентификатор 0, с помощью операционной системы считывает с диска первую половину вектора a и сохраняет данные в своей памяти под этим же именем. Затем она считывает вторую половину вектора a и помещает

информацию в буфер памяти temparray. Далее она отсылает сообщение с данными из этого буфера процессору, выполняющему программу с идентификатором 1. Потом те же действия выполняются для данных, составляющих вектор b. Программа с идентификатором 1 получает вторую половину векторов a и b и сохраняет таковые в своей памяти под их собственными именами.

```
integer array a[1...N/2], b[1...N/2], temparray[1...N/2]
integer dot_product
integer id
integer temp
...
id := mypid()
if   (id = 0) then
        read a[1...N/2] from vector_a
        read temparray[1...N/2] from vector_a
        send (temparray[1...N/2],1)
        read b[1...N/2] from vector_b
        read temparray[1...N/2] from vector_b
        send (temparray[1...N/2],1)
   else receive (a[1...N/2], 0)
        receive (b[1...N/2], 0)
end
dot_product :=0
do_dot(a, b)
if (id=0) send (dot_product, 0)
   else receive (temp, 1)
        dot_product := dot_product + temp
        print  dot_product
end
...
do_dot (integer array x[1...N/2], integer array y[1...N/2])
   for k := 1 to N/2
        dot_product := dot_product + x[k]*y[k]
   end
end
```

Рис. 12.20. Программа с передачей сообщений для вычисления скалярного произведения векторов на двух процессорах

Далее процедура do_dot просто вычисляет скалярное произведение N/2 элементов. Граничные значения цикла для обоих процессоров одинаковы, так как каждый использует данные из собственной памяти. Функция передачи сообщений иллюстрируется еще одним действием, выполняемым, когда процессор завершает процедуру do_dot. Чтобы программа с идентификатором 0 могла вычислить и напечатать результирующее скалярное произведение, программа с идентификатором 1 должна прислать ей свое скалярное произведение. Программа с идентификатором 0 принимает его во временный буфер temp.

Этот пример, как и предыдущий, нетрудно распространить на большее количество процессоров. Векторы должны быть разделены на части, назначаемые разным процессорам. Один из процессоров, например тот, который выполняет программу с идентификатором 0, назначается для вычисления конечного результата на основе данных, полученных от других процессоров.

Затраты на организацию параллельного выполнения программы несколькими процессорами включают время, необходимое для загрузки копий программ разными процессорами, время на загрузку частей вектора в память этих процессоров и время, требуемое на передачу между процессорами остальных сообщений. Поэтому скорость выполнения программы при ее распараллеливании между несколькими процессорами повышается только в том случае, если длина вектора достаточно велика и количество процессоров подобрано оптимальным образом.

У систем с общей памятью и передачей сообщений имеются свои сильные и слабые стороны. Системы с общей памятью использовать проще, поскольку они являются непосредственным расширением однопроцессорной архитектуры. Однако если данные располагаются в удаленных модулях памяти, задержки на обращение к этой памяти могут быть достаточно заметными, поэтому важно минимизировать количество операций доступа к глобальным переменным. Если объем сетевого трафика значительно повысится, сеть, скорее всего, начнет хуже функционировать. Производительность приложения в значительной степени определяется тем, насколько корректно программист выполнил синхронизацию.

Написание программ для систем с передачей сообщений — задача непростая, поскольку каждый процессор имеет собственное адресное пространство. В таких системах на передачу сообщений уходит много времени, и программист должен стараться так структурировать программы, чтобы предельно сократить эти затраты. А вот скорость работы сети обмена едва ли сильно отразится на производительности, так как сообщения передаются относительно редко. Синхронизация процессов выполняется неявно с помощью сообщений. Пожалуй, самым большим преимуществом систем с передачей сообщений является то, что они могут создаваться на базе более доступного и распространенного аппаратного обеспечения.

12.9. Производительность мультипроцессорных систем

Эта глава посвящена архитектуре систем, в которых время выполнения больших приложений сокращается за счет использования более чем одного процессора. Самой важной характеристикой производительности такой системы является параметр, определяющий повышение быстродействия работы приложений по сравнению с однопроцессорной системой. *Коэффициент ускорения* определяется как

$$S_p = \frac{T_1}{T_p}$$

где T_1 и T_P — время, необходимое для выполнения конкретного приложения при использовании одного и P процессоров соответственно. На рис. 12.21 показаны

три возможных варианта коэффициента ускорения в виде функций от количества процессоров в системе. Естественно предположить, что время выполнения программы должно сокращаться пропорционально увеличению числа процессоров. Такой коэффициент линейного ускорения, выражаемый равенством $S = P$, является идеальным для масштабируемых систем, однако его очень трудно достичь на практике.

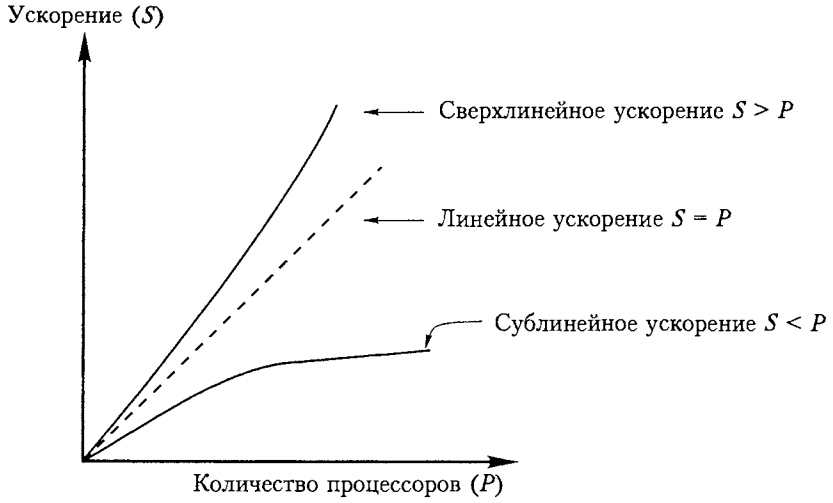


Рис. 12.21. Диаграмма изменения ускорения в мультипроцессорных системах

Как следует из предыдущего раздела, не все части программы можно распараллелить для выполнения на нескольких процессорах. Поэтому в ней останутся последовательные фрагменты, время выполнения которых не будет зависеть от количества используемых процессоров. Наличие таких фрагментов и их число в сравнении с общей длиной кода программы ограничивают реально достижимое ускорение.

Немаловажными причинами, по которым так трудно достичь линейного ускорения, являются затраты на инициализацию, синхронизацию, взаимодействие между процессами и согласование кэш-памяти, а также несбалансированность нагрузки. С увеличением размера системы эти затраты также повышаются. Все указанные факторы, за исключением несбалансированности нагрузки, мы обсуждали в предыдущих разделах. Как правило, перед переходом к выполнению очередной группы задач параллельной системе приходится ждать, пока последний процессор завершит текущую задачу. Поэтому при выполнении задачи с большим количеством процессоров важно, чтобы все они достигали точки синхронизации одновременно — тогда нагрузка на процессоры будет сбалансирована.

Коэффициент ускорения для большинства реальных приложений является сублинейной величиной, и начиная с определенного количества процессоров дальнейшего повышения производительности не происходит. Но иногда встречаются приложения, для которых возможно даже сверхлинейное ускорение. В следующем разделе мы приведем пример одного из таких приложений.

12.9.1. Закон Амдала

Рассмотрим параметры, влияющие на производительность мультипроцессорной системы. При усовершенствовании компьютерной системы всегда улучшается какая-то ее часть, но не вся система. Достигаемая при этом степень повышения производительности зависит от влияния «улучшенной» части на работу системы в целом. Данная идея сформулирована Джином Амдалом (Gene Amdahl) и известна как закон Амдала:

$$S_{new} = \frac{Старое\ время}{Новое\ время} = \frac{1}{1 - f_{enhanced} + f_{enhanced}/S_{enhanced}}$$

где

S_{new} — коэффициент ускорения в новой системе с внесенным усовершенствованием;

$S_{enhanced}$ — коэффициент ускорения при использовании только измененной части системы;

$f_{enhanced}$ — часть времени вычислений в старой системе, которая сокращается благодаря усовершенствованию.

Для мультипроцессорных систем приведенный закон можно сформулировать иначе. Пусть f — это та часть вычислений (в терминах времени), которую можно выполнять в параллельном режиме, P — количество процессоров в системе, а S_P — коэффициент ускорения по сравнению с последовательным выполнением. Тогда формулу для вычисления коэффициента ускорения можно записать так:

$$S_P = \frac{1}{1 - f + f/P} = \frac{P}{P - f(P-1)}$$

В этой формуле предполагается, что параллельная часть программы выполняется всеми процессорами при идеально сбалансированной нагрузке.

Предположим, что некое приложение запускается в 64-процессорной системе и 70 % его кода может выполняться в параллельном режиме. Тогда величину ожидаемого улучшения можно рассчитать следующим образом:

$$S_{64} = \frac{64}{64 - 0{,}7 \times 63} = 3{,}22$$

Если то же приложение запускается в 16-процессорной системе, ожидаемый коэффициент ускорения становится равным 2,91. Очевидно, что коэффициент ускорения получается много меньшим, чем можно было ожидать при увеличении количества процессоров. Так что нет никакого смысла использовать мощные мультипроцессоры для выполнения приложений, в которых имеются большие последовательные (не поддающиеся распараллеливанию) части. Добиться значительного ускорения можно лишь в том случае, когда последовательные части являются очень маленькими. Например, для приложения, в котором $f = 0{,}95$, коэффициент ускорения в описанных выше системах составляет 15,42 и 9,14

соответственно. Закон Амдала, по сути, утверждает, что линейное ускорение недостижимо, поскольку почти во всех приложениях имеются последовательные фрагменты.

До сих пор мы предполагали, что каждый процессор выполняет равную часть параллельных вычислений. Однако нагрузка не обязательно будет так идеально сбалансирована. Если перед переходом к следующему шагу приходится ждать, пока самый медленный процессор закончит свою часть работы, то результаты будут еще хуже, чем в приведенной выше формуле. Однако существуют и такие приложения, где задачи, выполняемые всеми процессорами, могут быть прекращены, как только один из процессоров завершит свою работу. Например, такое необычное поведение свойственно приложениям, основанным на алгоритме модельной «закалки» (улучшающей свойства модели). Предположим, что при разработке СБИС нужно так разместить на ней логические вентили, чтобы общая длина проводников в результирующей схеме получилась минимальной. Для этого необходимо сравнить большое количество разных вариантов размещения. С этой целью наиболее удачное из известных на данный момент размещений назначается всем процессорам в качестве начальной точки для следующей итерации. Далее каждый процессор может использовать свой случайный подход для изменения расположения вентилей в поисках наиболее удачной конфигурации. Как только один из процессоров определит конфигурацию, которая окажется лучше начальной, все вычисления будут остановлены и информация о ней будет передана всем остальным процессорам в качестве начального приближения для следующей итерации. Приложения подобного типа могут показать сверхлинейное ускорение, поскольку в случае их выполнения одним процессором последний может исследовать множество бесперспективных вариантов, пока не найдет хоть один подходящий.

12.9.2. Показатели производительности

С точки зрения пользователя самыми важными характеристиками компьютерной системы являются ее стоимость, удобство использования, надежность и производительность. Вычислительная способность компьютеров характеризуется несколькими показателями. Все, о чем рассказывалось в разделе 8.8, в равной степени применимо и к мультипроцессорным системам.

Производительность процессора можно определить как количество операций, выполняемых за одну секунду. Самыми популярными единицами измерения этого показателя являются MIPS (Million Instructions Per Second — миллионов команд в секунду) и MFLOPS (Million Floating Point Operations Per Second — миллионов операций с плавающей запятой в секунду, мегафлоп/с). Показатели MIPS и MFLOPS, приводимые производителем системы для конкретного процессора, характеризуют его максимальные возможности. Однако эти максимальные значения не всегда достижимы на практике. В мультипроцессорной системе показатели MIPS и MFLOPS являются просто суммами соответствующих показателей всех процессоров.

Еще один распространенный показатель производительности характеризует коммуникационные возможности сети, обычно определяемые как общая полоса пропускания (в байтах в секунду). При этом речь идет об оптимальной ситуации, когда по сети одновременно передается максимально возможное количество данных и занято максимально возможное число сетевых соединений.

Хотя такие показатели, как MIPS, MFLOPS и полоса пропускания, характеризуют возможности системы, они не отражают ее реальную производительность. В реальной работе используется только часть всех имеющихся в системе ресурсов, количество которых варьируется от системы к системе и от приложения к приложению. Корректное сравнение двух разных систем возможно только при выполнении в них конкретного набора приложений и измерении реальной производительности. Для осуществления такого сравнения разработано множество эталонных тестовых программ (benchmark program), отражающих поведение типичных приложений разного класса. Сравнение производительности систем при помощи эталонных тестовых программ стало широко распространенной практикой.

12.10. Резюме

Мультипроцессорные системы — это суперкомпьютерные системы, конфигурация которых может быть подобрана в соответствии с требованиями потребителя исходя из оптимального соотношения стоимости и производительности. Наилучшее соотношение этих показателей имеют системы, содержащие от десятков до сотен процессоров. Очень большие системы, состоящие из тысяч процессоров, трудно использовать на полную мощность, а их стоимость слишком высока.

С точки зрения стоимости достаточно просто реализовать мультикомпьютерную систему на основе рабочих станций, соединенных в локальную сеть. Эта возможность становится еще более привлекательной с повышением быстродействия локальных сетей.

Успешная работа мультикомпьютерной системы в значительной степени определяется наличием системного программного обеспечения, способного эффективно использовать имеющиеся ресурсы. Прикладная программа не будет выполняться быстро, если заложенные в ней возможности параллелизма и локализации не задействованы в полной мере. Компилятор должен выявить все возможности параллельного выполнения, а операционная система должна так распланировать процесс их реализации с учетом существующей локализации, чтобы те параллельные фрагменты кода, которые наиболее тесно связаны между собой и будут больше всего обмениваться информацией, выполнялись близко расположенными друг к другу процессорами. Для этого программист может предоставить некоторую вспомогательную информацию, но еще лучше, если системное программное обеспечение способно сделать это самостоятельно.

В настоящей главе были освещены наиболее важные аспекты мультипроцессорных и мультикомпьютерных систем. Для полного понимания возможностей и архитектуры указанных систем необходимо изучить их более детально.

Упражнения

12.1. Напишите программный цикл, команды которого могут широковещательно передаваться от управляющего процессора (рис. 12.1) для итерационного вычисления матричным процессором температуры в заданных точках проводящего слоя для примера из раздела 12.2. Кроме команд сдвига содержимого сетевых регистров между соседними процессорными элементами в нашей задаче имеются команды с двумя операндами, перемещающие данные между регистрами процессорных элементов и локальной памятью, а также команды для выполнения арифметических операций. Каждый процессорный элемент хранит текущее оценочное значение температуры в соответствующей точке своей локальной памяти по адресу CURRENT.

Для обработки можно использовать несколько регистров: R0, R1 и т. д. Процессорные элементы по краям матрицы содержат в своих сетевых регистрах фиксированные значения температуры и не участвуют в выполнении широковещательной программы. Значение, хранящееся по адресу EPSILON в каждом процессорном элементе, используется для определения того, достигнута ли заданная степень точности при вычислении значения локальной температуры. Если разность между вновь вычисленными значениями и содержиым ячейки CURRENT меньше заданной, то в конце каждой итерации цикла бит состояния STATUS должен устанавливаться в 1. В противном случае он устанавливается в 0.

12.2. Предположим, что передача данных по шине занимает T с, а время доступа к памяти равно $4T$ с. Тогда для выполнения запроса на чтение через обычную шину потребуется $6T$ с. Сколько шин нужно для того, чтобы полоса пропускания получилась такой же, как у шины с разделением транзакций, работающей с аналогичными временными задержками, или большей, чем у нее? Рассмотрите только запросы на чтение, игнорируя конфликты при обращении к памяти и предполагая, что все модули памяти соединены со всеми шинами. Увеличится или уменьшится это значение, если возрастет время доступа к памяти?

12.3. В мультипроцессоре с шинной топологией системная шина, не обеспечивающая достаточно высокую скорость передачи данных, может стать узким местом системы. Допустим, в системе используется шина с разделением транзакций, ширина которой в четыре раза превышает длину слова процессора. Будет ли при этом реальная скорость передачи данных вчетверо большей, чем при использовании такой же шины шириной в одно процессорное слово? Поясните свой ответ.

12.4. Предположим, что стоимость переключателя 2×2 в сети типа shuffle вдвое превышает стоимость точек соединения в сети с координатной коммутацией. В координатном коммутаторе $n \times n$ имеется n^2 точек соединения. При увеличении значения n сеть с координатной коммутацией становится дороже shuffle-сети. Каково наименьшее значение n, для которого сеть с координатной коммутацией будет дороже shuffle-сети в пять раз?

12.5. Сети типа shuffle можно создавать не только на основе переключателей 2×2, но и на основе переключателей большей размерности, например 4×4 или 8×8. Нарисуйте shuffle-сети 16×16 ($n = 16$) на основе переключателей 4×4. Если переключатель 4×4 стоит вчетверо больше переключателя 2×2, сравните стоимости сетей на их основе для значений n из последовательности 4, 4^2, 4^3 и т. д. Определите вероятность блокировки этих двух структур shuffle-сетей.

12.6. Допустим, что для выполнения каждой процедуры в PAR-сегменте (рис. 12.14) требуется 1 единица времени. Программа состоит из трех последовательных сегментов. Для выполнения каждого из них нужно k единиц времени, и каждый должен запускаться на отдельном процессоре. Три последовательных сегмента разделяются двумя сегментами PAR, состоящими из k процедур, которые могут выполняться на отдельных процессорах. Выведите формулу коэффициента ускорения для этой программы, если она обрабатывается мультипроцессором с n процессорами. Предполагается, что $n \leq k$. Каково предельное значение коэффициента ускорения при большом значении k и $n = k$? Что говорит этот результат о влиянии последовательных сегментов на выполнение такой программы, где некоторые сегменты имеют высокую степень параллелизма?

12.7. Кратчайшее расстояние, проходимое сообщением в n-мерном гиперкубе, равняется одной передаче, а наибольшее расстояние составляет n пересылок. Среднее проходимое сообщением расстояние будет меньше или больше $(1 + n)/2$, при условии, что вероятные параметры пары источник/приемник одинаковы? Попробуйте обосновать свой ответ.

12.8. Задача, которая ждет освобождения переменной, заблокированной командой Test and Set в состоящем из двух команд цикле (рис. 12.15), зря тратит циклы шины, которые могли бы быть задействованы для вычислений. Предложите способ решения этой проблемы с использованием централизованной очереди ждущих задач, поддерживаемой операционной системой. Предполагается, что операционная система может вызываться пользовательской задачей и сама выбирает, какая из ожидающих в очереди задач будет выполняться следующей.

12.9. Каковы аргументы за и против для двух стратегий согласования кэш-памяти, заключающихся в обновлении измененных данных и их пометке как недостоверных?

12.10. В разделе 12.6.3 утверждалось, что схемы согласования кэша не избавляют от необходимости использовать переменные блокировки. А могут ли переменные блокировки заменить собой схемы согласования кэша?

12.11. Удастся ли добиться повышения производительности, если вместо программы, показанной на рис. 12.18, использовать программу с рис. 12.19. Сколько времени будет выполняться каждый шаг программы?

12.12. Модифицируйте программу на рис. 12.19 для выполнения в четырехпроцессорной системе.

12.13. Модифицируйте программу на рис. 12.20 для выполнения в четырехпроцессорной системе.

12.14. Для небольших векторов подход, проиллюстрированный на рис. 12.19, приводит к худшим результатам, чем выполнение программы на одном процессоре. Оцените минимальный размер вектора, начиная с которого применение такого подхода способствует ускорению работы программы. Сколько времени будет выполняться каждый шаг программы?

12.15. Повторите упражнение 12.14 для подхода, показанного на рис. 12.20.

12.16. Мультипроцессоры с общей памятью и мультикомпьютеры с передачей сообщений — это две архитектуры, поддерживающие параллельное выполнение взаимодействующих друг с другом задач. Для какой из них проще эмулировать работу другой архитектуры? Коротко поясните свой ответ.

12.17. Для локальных сетей, в которых время передачи сообщений значительно больше 2τ (где τ — это время пересылки сообщения от одного конца сети к другому), единственно подходящим протоколом является шинный протокол Ethernet. Рассмотрим случай, когда время передачи сообщения меньше τ. Может ли станция назначения получить неискаженное сообщение, если исходная станция обнаружит конфликт в течение периода окна конфликта продолжительностью 2τ? Аргументируйте свой ответ. Если вы думаете, что это возможно, укажите возможное расположение источника, приемника и конфликтующей станции на шине, а также время каждого из событий.

12.18. RAM-память с элементом BOXLOC называется *почтовым ящиком*. С каждым словом этой памяти связан бит F/E (полон/пуст). Команда

```
PUT  R0,BOXLOC,WAITREC
```

выполняется следующим образом. Сначала проверяется бит F/E, связанный с адресом BOXLOC в памяти почтового ящика. Если он равен 0, то есть ящик пуст, содержимое регистра R0 записывается в переменную BOXLOC, бит F/E устанавливается в 1 (это означает, что ящик полон) и выполнение продолжается со следующей команды. В противном случае (если F/E = 1) никакие действия не производятся и управление передается команде, расположенной в памяти программы по адресу WAITREC.

а) Приведите определение команды

```
GET  R0,BOXLOC,WAITREC
```

дополняющей команду PUT.

б) Предположим, что две задачи, T_1 и T_2, обрабатываются в мультипроцессорной системе на разных процессорах. От T_1 к T_2 при помощи команд PUT и GET передается поток сообщений длиной в одно слово, помещаемых в почтовый ящик в общей памяти. Напишите на унифицированном языке ассемблера фрагменты программ T_1 и T_2, выполняющих те же операции в мультипроцессорной системе с общей памятью, где нет почтового ящика, но есть команда TAS (см. раздел 12.6.1).

Приложение А

Логические схемы

В цифровых компьютерах информация представляется и обрабатывается с помощью электронных *логических схем*. Логические схемы оперируют двоичными переменными, принимающими одно из двух значений (обычно таковыми являются нуль и единица). В данном приложении вы познакомитесь с понятием логических функций и узнаете, как строятся реализующие их логические схемы. Здесь же приведен краткий обзор технологий создания логических схем.

А.1. Базовые логические функции

Рассказ о двоичной логике проще всего начать с простого примера, знакомого многим из вас. Представьте себе обычную электрическую лампочку, состояние которой (включена/выключена) управляется двумя выключателями, x_1 и x_2. Каждый из выключателей может находиться в одном из двух возможных положений, 0 или 1 (рис. А.1, *а*). Это означает, что его можно представить как двоичную переменную. Поэтому пусть имена переключателей служат и именами соответствующих им двоичных переменных. Еще на рисунке показаны источник питания и сама лампочка. То, как выключатели будут управлять включением и выключением лампочки, зависит от соединения их проводов. Свет горит лишь в том случае, если образуется замкнутый контур, соединяющий лампочку с источником питания. Пусть условие включения лампочки представляет двоичная переменная f. Если лампочка включена, значит, $f = 1$, а если она выключена, то $f = 0$. Таким образом, условие $f = 1$ указывает, что в цепи существует как минимум один замкнутый контур, а условие $f = 1$ означает, что замкнутого контура нет. Очевидно, что f является функцией двух переменных, x_1 и x_2.

Теперь давайте рассмотрим существующие способы управления лампочкой. Для начала предположим, что она будет гореть при условии, что хотя бы один из переключателей находится в положении 1, то есть $f = 1$, если

$$x_1 = 1 \quad \text{и} \quad x_2 = 0$$

или

$$x_1 = 0 \quad \text{и} \quad x_2 = 1$$

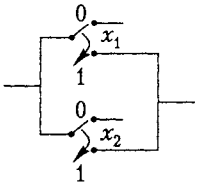
или

$$x_1 = 1 \quad \text{и} \quad x_2 = 1$$

Соединения, реализующие этот тип управления, показаны на рис. А.1, *б*. Рядом со схемой приведена представляющая эту ситуацию логическая *таблица истинности*. В таблице перечислены все возможные пары установок переключателей

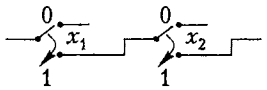
и соответствующие им значения функции f. В терминах математической логики эта таблица представляет функцию ИЛИ (OR) переменных x_1 и x_2.

а



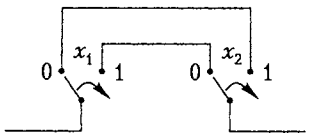
x_1	x_2	$f(x_1,x_2) = x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

б



x_1	x_2	$f(x_1,x_2) = x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

в



x_1	x_2	$f(x_1,x_2) = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

г

Рис. А.1. Схемы включения электрической лампочки: лампочка, управляемая двумя выключателями (*а*); параллельное соединение выключателей — схема ИЛИ (*б*); последовательное соединение выключателей — схема И (*в*); соединение выключателей по схеме Исключающее ИЛИ (*г*)

Операцию ИЛИ обычно представляют алгебраическим знаком «+» или «∨», так что

$$f = x_1 + x_2 = x_1 \vee x_2$$

Мы говорим, что x_1 и x_2 являются *входными* переменными, а f — это *выходная* функция.

Следует указать некоторые важнейшие свойства операции ИЛИ. Прежде всего, она коммутативна, то есть

$$x_1 + x_2 = x_2 + x_1$$

Данная операция может распространятся на n переменных, так что функция

$$f = x_1 + x_2 + \ldots + x_n$$

принимает значение 1, если это же значение имеет хотя бы одна переменная x_i. Проанализировав таблицу истинности, вы увидите, что

$$1 + x = 1$$

и

$$0 + x = x$$

А теперь предположим, что лампочка должна загораться только в том случае, если оба выключателя находятся в положении 1. Такая схема соединения выключателей с соответствующей ей таблицей истинности показана на рис. А.1, *в*. Эта схема соответствует функции И (AND), для обозначения которой используется символ «·» или «∧»:

$$f = x_1 \cdot x_2 = x_1 \wedge x_2$$

Вот важнейшие свойства операции И:

$$x_1 \cdot x_2 = x_2 \cdot x_1$$
$$1 \cdot x = x$$
$$0 \cdot x = 0$$

Функцию И тоже можно распространить на n переменных:

$$f = x_1 \cdot x_2 \cdot \ldots \cdot x_n$$

Эта функция имеет значение 1 только в том случае, если все переменные x_i имеют значение 1. Она представляет такую же схему, как на рис. А.1, *в*, в которой, правда, последовательно соединено большее количество выключателей.

Последний вариант соединения выключателей также достаточно распространен. Здесь выключатели подсоединены с двух концов ступенчатого контура, так что лампочку можно включать и выключать с помощью любого из них. Это означает, что если свет включен, изменением положения любого из выключателей его можно выключить, а если свет выключен, изменением положения любого из выключателей его можно включить. Предположим, что лампочка не горит, когда оба

выключателя находятся в положении 0. Переключение же любого из них в положение 1 включает лампочку. Теперь предположим, что лампочка горит, если $x_1 = 1$, а $x_2 = 0$. Переключение x_1 в положение 0 выключает лампочку. Более того, для ее выключения можно также установить x_2 в положение 1, то есть $f = 0$, если $x_1 = x_2 = 1$. Соединение, которое реализует этот способ управления лампочкой, показано на рис. А.1, *г*. Соответствующая логическая операция, представляемая символом «⊕», называется Исключающее ИЛИ (EXCLUSIVE-OR или XOR). Приведем ее важнейшие свойства:

$$x_1 \oplus x_2 = x_2 \oplus x_1$$
$$1 \oplus x = \bar{x}$$
$$0 \oplus x = x$$

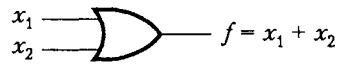
где $\bar{x}$ обозначает функцию НЕ (NOT) от переменной x. Эта функция переменной $f = \bar{x}$ имеет значение 1, если $x = 0$, и значение 0, если $x = 1$. В подобном случае мы говорим, что входное значение x *инвертируется* или *дополняется*.

А.1.1. Электронные логические вентили

Наш пример с выключателями, замкнутыми и разомкнутыми электрическими цепями и лампочками, иллюстрирующий идею логических переменных и функций, удобен тем, что он очень прост и каждому знаком. При этом представленные им логические концепции применимы к электрическим цепям, используемым для обработки информации в цифровых компьютерах. Физическими переменными в данном случае являются не положения выключателей и замкнутые или разомкнутые цепи, а электрическое напряжение и ток. Для примера рассмотрим схему, предназначенную для работы со входным напряжением +5 В или 0 В. Возможные значения выходного напряжения в ней тоже составляют +5 или 0 В. Если мы договоримся, что значение +5 В представляет логическую единицу, а значение 0 В — логический нуль, тогда функционирование этой схемы можно будет описать с помощью таблицы истинности той логической операции, которую она реализует.

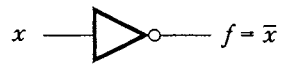
С применением транзисторов можно сконструировать простые электронные схемы, которые будут выполнять логические операции И, ИЛИ, Исключающее ИЛИ и НЕ. Эти базовые схемы традиционно называют *вентилями* (gates). Стандартные обозначения вентилей всех четырех типов приведены на рис. А.2. Если операция НЕ применяется ко входному или выходному значению логического вентиля, для нее используется упрощенное обозначение — просто маленький кружок.

Об электронной реализации логических вентилей говорится в разделе А.5. А пока мы с вами поговорим о том, как с помощью базовых вентилей конструируются логические схемы, реализующие более сложные логические функции.

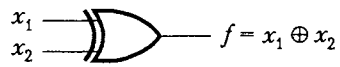


Вентиль ИЛИ

Вентиль И



Вентиль НЕ



Вентиль
Исключающее ИЛИ

Рис. А.2. Стандартные обозначения логических вентилей

А.2. Объединение логических функций

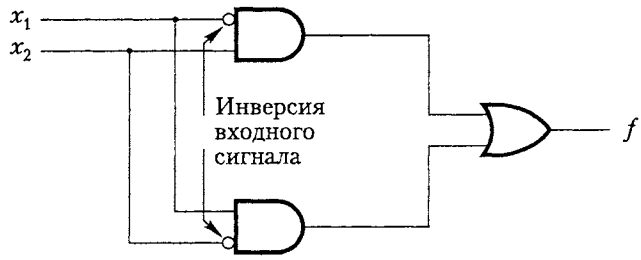
Рассмотрим схему, которая состоит из двух вентилей И и одного вентиля ИЛИ (рис. А.3, *а*). Она может быть представлена выражением

$$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

Схема составления таблицы истинности для этого выражения показана на рис. А.3, *б*. Сначала для каждого входного значения определяются значения термов И, затем, с помощью операции ИЛИ, — результирующие значения функции *f*. Таблица истинности функции *f* идентична таблице истинности функции Исключающее ИЛИ, так что схема с тремя вентилями, показанная на рис. А.3, *а*, реализует функцию Исключающее ИЛИ с помощью вентилей И, ИЛИ и НЕ. Логическое выражение $\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$ называется *суммой произведений*, поскольку операцию ИЛИ иногда называют суммой, а операцию И — произведением. Следует отметить, что правильнее было бы записать это выражение так:

$$f = ((\bar{x}_1) \cdot x_2) + (x_1 \cdot (\bar{x}_2))$$

Такая форма записи, как вы понимаете, отражает порядок применения логических операций. Для упрощения подобных выражений определяют иерархию операций И, ИЛИ и НЕ. Если в выражении отсутствуют скобки, логические операции выполняются в следующем порядке: сначала НЕ, затем И и только после этого ИЛИ. Более того, оператор «·» часто вовсе пропускают, если выражение не допускает двухзначной интерпретации.



$$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

а

x_1	x_2	$\bar{x}_1 \cdot x_2$	$x_1 \cdot \bar{x}_2$	$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2 = x_1 \oplus x_2$
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

б

Рис. А.3. Реализация функции Исключающее ИЛИ с использованием вентилей И, ИЛИ и НЕ: схема для функции Исключающее ИЛИ (*а*); таблица истинности для выражения $\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$ (*б*)

Возвращаясь к сумме произведений, мы сейчас покажем, как можно синтезировать любую логическую функцию непосредственно на основе ее таблицы истинности (табл. А.1).

Таблица А.1. Функции трех переменных

x_1	x_2	x_3	f_1	f_2
0	0	0	1	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

Предположим, мы хотим составить схему функции f_1 на основе вентилей И, ИЛИ и НЕ. Для каждой строки таблицы, в которой $f_1 = 1$, в формулу суммы произведений включается терм И со всеми тремя входными переменными. К одной, двум или трем из этих переменных по отдельности нужно применить оператор НЕ — таким образом, чтобы терм был равен 1 только в том случае, когда значения

переменных соответствуют данной строке таблицы истинности. Это означает, что если в этой строке $x_i = 0$, в произведение включается элемент $\bar{x}_i$, а если $x_i = 1$ — элемент x_i. Например, в четвертой строке таблицы истинности значение функции 1 соответствует входным значениям

$$(x_1, x_2, x_3) = (0, 1, 1)$$

Данной строке соответствует терм $\bar{x}_1 x_2 x_3$. Составив аналогичные термы для всех строк таблицы истинности, в которых функция f_1 имеет значение 1, мы получим вот такую сумму произведений:

$$f_1 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3$$

Логическая схема, соответствующая этому выражению, приведена в левой части рис. А.4. В качестве еще одного примера использования описанного алгоритма можно сформировать сумму произведений для функции Исключающее ИЛИ. Этот алгоритм может применяться с целью формирования суммы произведений и соответствующей логической схемы на основе таблицы истинности любого размера.

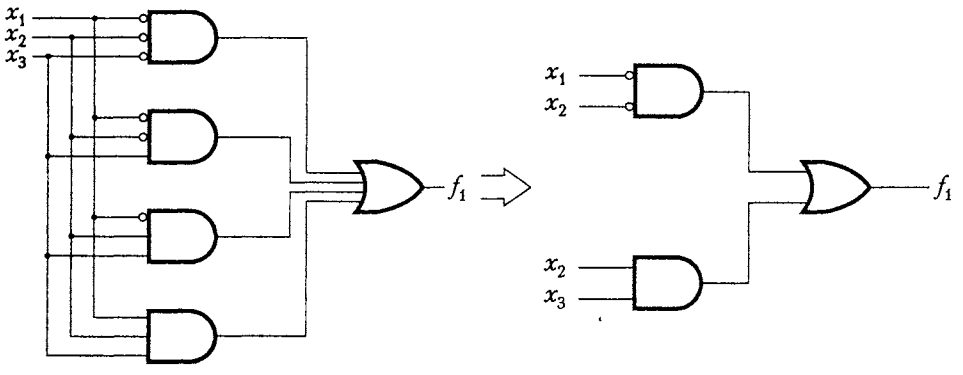


Рис. А.4. Логическая схема для функции f_1 из табл. А.1 и соответствующая ей минимальная схема реализации

А.3. Минимизация логических выражений

Теперь вы знаете, как формируется сумма произведений для произвольной таблицы истинности. Фактически для каждой таблицы истинности существует множество эквивалентных выражений и логических схем. Два логических выражения или две логические схемы считаются эквивалентными, если у них одинаковые таблицы истинности. Сформированной нами в предыдущем разделе сумме произведений для функции f_1 эквивалентно, в частности, такое выражение:

$$\bar{x}_1\bar{x}_2 + x_2 x_3$$

Чтобы это доказать, достаточно составить таблицу истинности данного выражения и сравнить ее с таблицей А.1. Процесс создания таблицы истинности выражения $\bar{x}_1\bar{x}_2 + x_2x_3$, приведенной в табл. А.2, можно разбить на три этапа. Сначала для каждого набора входных значений вычисляется произведение $\bar{x}_1\bar{x}_2$, затем — произведение x_2x_3, после чего оба результата складываются для получения окончательного значения. Как видите, наша таблица истинности идентична таблице истинности функции f_1, приведенной в табл. А.1.

Для упрощения логических выражений выполняется ряд алгебраических операций. Они основаны на двух логических законах, о которых мы еще не упоминали: дистрибутивном

$$w(y + z) = wy + wz$$

и законе исключенного третьего:

$$w + \bar{w} = 1$$

Таблица А.2. Вычисление выражения $\bar{x}_1\bar{x}_2 + x_2x_3$

x_1	x_2	x_3	$\bar{x}_1\bar{x}_2$	x_2x_3	$\bar{x}_1\bar{x}_2 + x_2x_3 = f_1$
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

Таблица А.3. Использование таблиц истинности для доказательства эквивалентности выражений

w	y	z	y + z	Значение $w(y + z)$	wy	wz	Значение $wy + wz$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

В табл. А.3 приведено доказательство истинности дистрибутивного закона. Очевидно, что подобные законы всегда можно доказать с помощью таблиц истинности выражений, стоящих справа и слева от знака равенства. Совпадение результирующих значений в этих двух таблицах подтверждает проверяемый закон. Логические законы, подобные дистрибутивному, иногда называют *тождествами*.

Вот еще одна форма дистрибутивного закона, которую мы приводим для полноты изложения материала, хотя она нам и не потребуется:

$$w + yz = (w+y)(w + z)$$

Целью минимизации логического выражения, представляющего заданную логическую функцию, является уменьшение стоимости ее реализации (количества используемых логических элементов). Общая схема процесса реализации логической функции такова. Сначала по описанному нами алгоритму для нее составляется сумма произведений (дизъюнктивная совершенная нормальная форма). Затем полученное выражение минимизируют до эквивалентной *минимальной суммы произведений*. Чтобы определить критерий минимизации, нужно ввести понятие стоимости, или величины, логического выражения.

Обычно при оценке стоимости выражения учитывается общее количество вентилей и их входных значений (входных линий), необходимых для реализации выражения в форме, показанной на рис. А.4. Например, стоимость большей схемы на этом рисунке равна 21: 5 вентилей плюс 16 входных значений. Инверсия входных значений при подсчете игнорируется. Стоимость более простого выражения равна 9: 3 вентиля плюс 6 входных значений. Теперь можно определить и критерий минимизации. Сумма произведений считается минимальной, если не существует эквивалентного ей выражения меньшей стоимости. В простых примерах, которые мы будем рассматривать в этой книге, минимальный размер выражений будет очевидным. Поэтому мы не считаем нужным приводить строгие доказательства их минимальности.

Стратегия упрощения заданного выражения заключается в следующем. Прежде всего термы-произведения разбиваются на пары, отличающиеся единой переменной, которая в одном терме дополняется ($\bar{x}$), а во втором используется как есть (x). Затем в каждой паре общее произведение двух переменных выносится за скобки, а в скобках остается терм $x + \bar{x}$, всегда равный 1. Вот что мы получим, применив эту процедуру к первому выражению для функции f_1:

$$\begin{aligned} f_1 &= \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2x_3 + x_1x_2x_3 \\ &= \bar{x}_1\bar{x}_2(\sim x_3 + x_3) + (\bar{x}_1 + x_1)x_2x_3 \\ &= \bar{x}_1\bar{x}_2 \cdot 1 + 1 \cdot x_2x_3 \\ &= \bar{x}_1\bar{x}_2 + x_2x_3 \end{aligned}$$

Это выражение минимально. Соответствующая ему логическая схема приведена на уже упоминаемом нами рис. А.4.

Сгруппировать термы попарно, с тем чтобы упростить исходное выражение, не всегда так просто, как в примере с функцией f_1. В случае затруднений помогает такое правило:

$$w + w = w$$

Это правило позволяет повторять термы-произведения при необходимости объединить некоторый терм более чем с одним другим термом. Для примера рассмотрим функцию f_2 из табл. А.1. Исходная сума произведений, формируемая на основе таблицы истинности этой функции, такова:

$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \sim x_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3$$

Повторив первый терм $\sim x_1 \sim x_2 \sim x_3$ и изменив порядок следования термов (на основе коммутативного закона), мы получим:

$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3$$

Сгруппировав термы попарно и вынеся одинаковые произведения за скобки, мы сможем записать следующее выражение:

$$\begin{aligned} f_2 &= \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) + x_1\bar{x}_2(\bar{x}_3 + x_3) + \bar{x}_1(\bar{x}_2 + x_2)\bar{x}_3 = \\ &= \bar{x}_1\bar{x}_2 + x_1\bar{x}_2 + \bar{x}_1\bar{x}_3 \end{aligned}$$

Первую пару термов можно упростить еще раз, и тогда получится минимальное выражение:

$$f_2 = \bar{x}_2 + \bar{x}_1\bar{x}_3$$

На этом обсуждение способов алгебраического упрощения логических выражений завершается. Данное математическое упражнение еще раз доказывает, что более простые логические схемы, содержащие меньше вентилей и входных значений, легче и дешевле реализовать на практике. Так что стремление минимизировать логические выражения имеет под собой чисто экономическое основание. Законы, которые мы с вами использовали для манипулирования логическими выражениями, объединены в табл. А.4. Они приведены парами, чтобы видна была симметрия функций И и ИЛИ. До сих пор нам не представилось случая воспользоваться законами возведения в степень и де Моргана, но в следующих разделах они нам пригодятся.

Таблица А.4. Законы двоичной логики

Название закона	Алгебраическое тождество	
Коммутативный	$w + y = y + w$	$wy = yw$
Ассоциативный	$(w + y) + z = y + (w + z)$	$(wy)z = w(yz)$
Дистрибутивный	$w + yz = (w + y)(w + z)$	$w(y + z) = wy + wz$
Идемпотентности	$w + w = w$	$ww = w$
Возведение в степень	$\overline{\overline{w}} = w$	
Дополнения (закон исключенного третьего)	$w + \overline{w} = 1$	$w\overline{w} = 0$
Закон де Моргана	$\overline{(w + y)} = \overline{w}\,\overline{y}$	$\overline{wy} = \overline{w} + \overline{y}$
	$1 + w = 1$	$0 \cdot w = 0$
	$0 + w = w$	$1 \cdot w = w$

А.3.1. Минимизация функций с использованием карты Карно

При минимизации функций f_1 и f_2 из табл. А.1 нам приходилось искать наиболее эффективные способы преобразования исходных выражений. Например, далеко

не очевидным было решение повторить терм $\bar{x}_1\bar{x}_2\bar{x}_3$ на первом шаге минимизации функции f_2. Для того чтобы как можно быстрее получить минимальное выражение, представляющее логическую функцию нескольких переменных, можно воспользоваться графическим представлением таблицы истинности, называемым *картой Карно*. Для функции трех переменных карта Карно представляет собой прямоугольник, составленный из восьми квадратов, расположенных в два ряда по четыре в каждом (рис. А.5, *а*). Каждый квадрат соответствует конкретному набору значений входных переменных. Например, третий квадрат в верхнем ряду представляет значения $(x_1, x_2, x_3) = (1, 1, 0)$. Поскольку в таблице истинности функции трех переменных содержится восемь строк, карта должна состоять из восьми квадратов. Значения внутри квадратов — это значения функции при соответствующих значениях переменных.

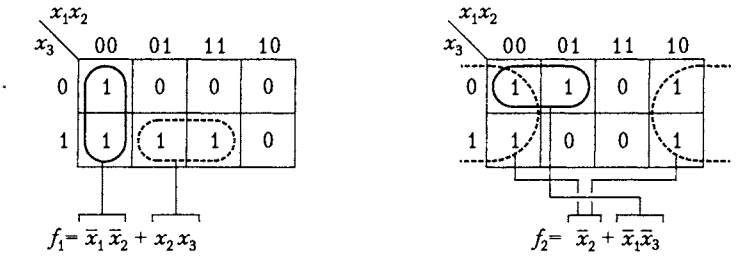
Главная идея карты Карно заключается в том, что расположенные рядом по горизонтали и по вертикали квадраты отличаются значениями только одной переменной. Если два смежных квадрата содержат единицы, это означает возможность алгебраического упрощения соответствующей пары термов. Например, на карте функции f_2 (рис. А.5, *а*) единицы в двух крайних слева квадратах верхнего ряда соответствуют термам $\bar{x}_1\bar{x}_2\bar{x}_3$ и $\bar{x}_1x_2\bar{x}_3$. Эта пара термов упрощается следующим образом:

$$\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 = \bar{x}_1\bar{x}_3$$

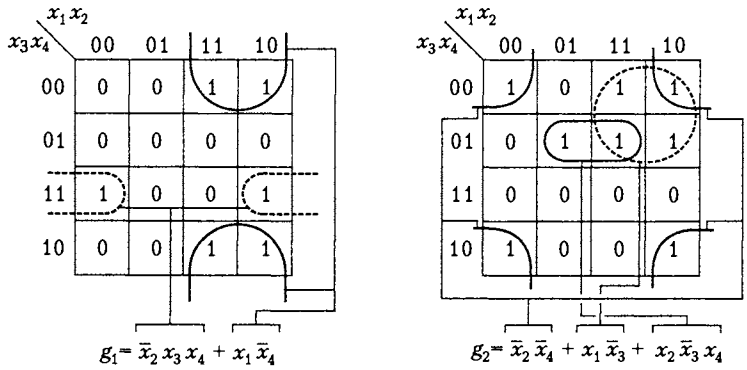
что мы и сделали в предыдущем разделе при минимизации алгебраического выражения для функции f_2. Минимизированное произведение, соответствующее группе квадратов, — это произведение входных переменных, значения которых одинаковы для всех квадратов этой группы. Если значение входной переменной x_i равно нулю для всех квадратов группы, тогда переменная x_i входит в результирующее произведение. Квадраты с левого края карты считаются смежными с квадратами с ее правого края. Так, в карте функции f_2 имеется группа из четырех единиц, состоящая из крайнего слева столбца и крайнего справа столбца карты. Соответствующая группа термов упрощается до одного терма $\bar{x}_2$, содержащего единственную переменную, поскольку только переменная x_2 имеет одинаковые значения во всех квадратах группы.

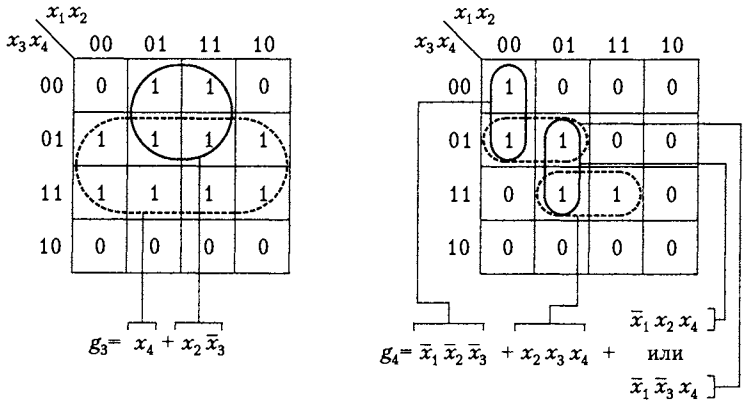
Карты Карно могут использоваться и для минимизации функций более чем трех переменных. Карту для четырех переменных можно составить из двух карт для трех переменных. Два примера таких карт показаны на рис. А.5, *б*, и под каждой из них приведено минимальное выражение для представляемой ею функции. Если на карте для трех переменных квадраты можно группировать по два и по четыре, то на карте для четырех переменных их можно группировать еще и по восемь. Пример такой группировки показан на карте функции g_3. Обратите внимание, что четыре угловых квадрата можно объединить в одну группу, как на карте функции g_2, где на их основе составлен терм $\bar{x}_2\bar{x}_4$. Как и в случае карты для трех переменных, терм, соответствующий группе квадратов, представляет собой произведение переменных, значения которых одинаковы для всех квадратов этой группы. Так, в группе из четырех квадратов в правом верхнем углу карты функции g_2 во всех квадратах $x_1 = 1$ и $x_3 = 0$, поэтому эту группу представляет терм $x_1\bar{x}_3$. Остальные две переменные, x_2 и x_4, имеют в квадратах этой группы разные

значения. Карты Карно можно использовать и для функций пяти переменных. В этом случае для представления функции используются две карты для четырех переменных: одна из них соответствует значению 0 пятой переменной, а другая — ее значению 1.



а





б

Рис. А.5. Минимизация функций с использованием карт Карно: карта для трех переменных (*а*); карта для четырех переменных (*б*)

Общая процедура формирования на карте Карно групп из двух, четырех, восьми и т. д. квадратов определяется просто. Две смежные пары квадратов, содержащих единицы, можно объединить в группу из четырех квадратов. Две смежные группы по четыре квадрата можно объединить в группу из восьми квадратов. В общем случае количество квадратов в группе должно быть равным 2^k, где k — целое число.

Теперь давайте рассмотрим процедуру получения с помощью карты Карно минимальной суммы произведений. Как видно на рис. 2.5, большей группе квадратов соответствует произведение меньшего числа переменных. Поэтому для получения минимального выражения нужно объединить все квадраты на карте, содержащие единицы, в как можно меньшее количество групп, выбирая наибольшие из них, так чтобы при этом охватить все единицы. Для примера рассмотрим карту функции g_2, приведенную на рис. А.5, *б*. Как вы уже знаете, единицы по ее углам составляют группу из четырех квадратов, представляемую термом $\bar{x}_2\bar{x}_4$. Еще одна группа из четырех квадратов располагается в правом верхнем углу и представлена термом $x_1\bar{x}_3$. Эти две группы охватывают все единицы на карте, за исключением одной единицы в квадрате $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$. Наибольшая группа единиц, включающая этот квадрат, — это группа из двух квадратов, представленная термом $x_2\bar{x}_3x_4$. Таким образом, минимальное выражение для функции g_2 должно быть следующим:

$$g_2 = \bar{x}_2\bar{x}_4 + x_1\bar{x}_3 + x_2\bar{x}_3x_4$$

Минимальные выражения для других функций, представленных на рис. А.5, *б*, формируются аналогичным образом. Обратите внимание, что в случае функции g_4 существует два альтернативных выражения: одно включает терм $\bar{x}_1x_2x_4$, а второе — терм $\bar{x}_1\bar{x}_3x_4$. Такое случается довольно часто.

Во всех наших примерах составить минимальное выражение достаточно просто. Вообще-то для этой цели существуют формальные алгоритмы, но мы их рассматривать не будем.

А.3.2. Безразличные значения

В некоторых случаях определенные наборы входных значений цифровых схем никогда не используются. Для примера рассмотрим двоично-десятичное представление числа (Binary-Coded Decimal, BCD). Десятичные цифры от 0 до 9 можно представить с помощью четырех двоичных переменных, b_3, b_2, b_1 и b_0 (рис. А.6). Эти четыре переменные могут составить 16 разных наборов значений, из которых для представления десятичных цифр используются только 10. Оставшиеся значения не используются. Следовательно, логическая схема, обрабатывающая данные в формате BCD, никогда не получит в качестве входных данных ни один из шести оставшихся наборов значений.

На рис. А.6 приведена таблица истинности для конкретной функции, принимающей в качестве аргумента двоично-кодированную десятичную цифру. Значения этой функции для неиспользуемых входных наборов нас, естественно, не интересуют. Такие значения называются *безразличными* (don't care) и в таблице истинности они обозначаются буквой «d». При реализации функции им можно присвоить либо нуль, либо единицу, в зависимости от того, какое из этих двух

значений позволит минимизировать результирующую схему. Единица присваивается в том случае, если такая замена приводит к расширению группы ячеек с единичными значениями функции. Поскольку большим группам соответствуют меньшие выражения, результат лучше минимизируется.

Представляемая десятичная цифра	Двоичный код b_3	b_2	b_1	b_0	f
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
Не используются	1	0	1	0	d
	1	0	1	1	d
	1	1	0	0	d
	1	1	0	1	d
	1	1	1	0	d
	1	1	1	1	d

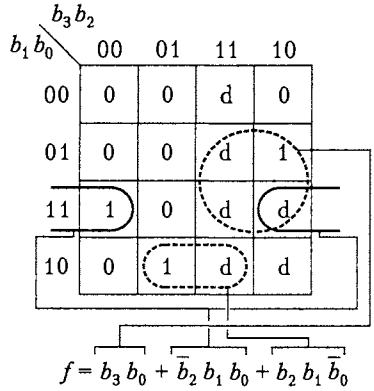


Рис. А.6. Карта Карно для четырех логических переменных

Функция, приведенная на рис. А.6, реализует следующий алгоритм обработки входной десятичной цифры: выходное значение равно 1, если входное значение

является любым ненулевым числом, кратным 3. Три единицы на карте Карно расположены таким образом, что для их охвата требуются три группы квадратов, а безразличные значения определяются так, чтобы предельно увеличить размеры этих групп.

А.4. Синтез вентилей И-НЕ и ИЛИ-НЕ

А сейчас нам предстоит рассмотреть еще два базовых логических вентиля, называемых И-НЕ и ИЛИ-НЕ. Эти вентили очень широко применяются в логических схемах, что объясняется простотой их реализации. Таблицы истинности вентилей И-НЕ и ИЛИ-НЕ приведены на рис. А.7. Они представляют собой функции И и ИЛИ, к результату которых применена функция НЕ. Если мы обозначим операторы И-НЕ и ИЛИ-НЕ символами «↑» и «↓», то, используя закон де Моргана (см. табл. А.4), сможем представить их следующим образом:

$$x_1 \uparrow x_2 = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$$

$$x_1 \downarrow x_2 = \overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$$

Вентили И-НЕ и ИЛИ-НЕ могут использоваться и с более чем двумя входными переменными, и действуют они в соответствии с очевидным обобщением закона де Моргана:

$$x_1 \uparrow x_2 \uparrow \ldots \uparrow x_n = \overline{x_1 x_2 \ldots x_n} = \bar{x}_1 + \bar{x}_2 + \ldots + \bar{x}_n$$

и

$$x_1 \downarrow x_2 \downarrow \ldots \downarrow x_n = \overline{x_1 + x_2 + \ldots + x_n} = \bar{x}_1 \bar{x}_2 \ldots \bar{x}_n$$

Разработка логических схем с вентилями И-НЕ и ИЛИ-НЕ не так проста, как разработка схем с вентилями И, ИЛИ и НЕ. Одной из главных трудностей в решении этой задачи является то, что по отношению к операциям И-НЕ и ИЛИ-НЕ ассоциативный закон не действует. Позже мы с вами еще вернемся к этой проблеме. Но сейчас давайте рассмотрим простую и универсальную процедуру синтеза произвольной логической функции с использованием только вентилей И-НЕ. Эту процедуру легче всего продемонстрировать на примере. Произведем алгебраическое преобразование логического выражения, соответствующего схеме с четырьмя входными переменными, которая включает три вентиля И-НЕ, имеющих по две входные переменные:

$$\begin{aligned}(x_1 \uparrow x_2) \uparrow (x_3 \uparrow x_4) &= \overline{\overline{(x_1 x_2)}\,\overline{(x_1 x_2)}} \\ &= \overline{\overline{x_1 x_2}} + \overline{\overline{x_3 x_4}} \\ &= x_1 x_2 + x_3 x_4\end{aligned}$$

Для выполнения нужных преобразований мы воспользовались законами де Моргана и возведения в степень. На рис. А.8 показаны логические схемы, соответствующие этим преобразованиям. Поскольку любую логическую функцию

можно синтезировать с помощью суммы произведений (И-ИЛИ) и поскольку приведенные преобразования обратимы, мы можем сделать вывод, что любую логическую функцию можно синтезировать в форме И-НЕ–И-НЕ. Причем это предположение верно для функций с любым количеством переменных. Очевидно, что общее количество входных переменных вентилей И-НЕ при этом должно быть таким же, как общее количество входных переменных вентилей И и ИЛИ.

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	0

$f = x_1 \uparrow x_2 = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$ $\qquad$ $f = x_1 \downarrow x_2 = \overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

а $\qquad$ *б*

Рис. А.7. Вентили: И-НЕ (*а*); ИЛИ-НЕ (*б*)

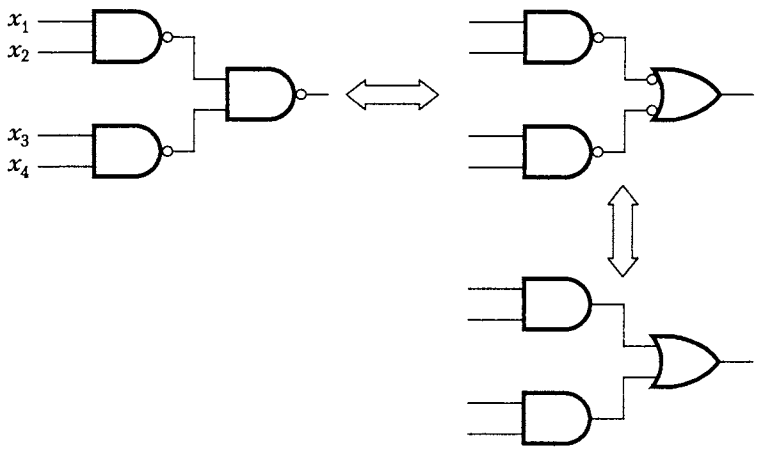


Рис. А.8. Эквивалентные схемы на основе вентилей И-НЕ и вентилей И и ИЛИ

Теперь давайте вернемся к вопросу о затруднении, возникающем вследствие того, что оператор И-НЕ не подчиняется ассоциативному закону. При разработке логической схемы на основе вентилей И-НЕ с использованием процедуры, иллюстрируемой рис. А.8, может потребоваться вентиль И-НЕ с большим количеством

входов, чем можно реализовать физически. Для схемы на основе вентилей И и ИЛИ это не проблема, поскольку благодаря ассоциативности данных операций один вентиль с большим количеством входов можно заменить несколькими вентилями с меньшим количеством входов, как на рис. А.9, *а*. В случае вентиля И-НЕ решение не так просто. Например, функцию И-НЕ с тремя входными переменными нельзя реализовать в виде соединения двух вентилей И-НЕ, имеющих по два входа. Как показано на рис. А.9, *б*, для ее реализации потребуется три вентиля И-НЕ.

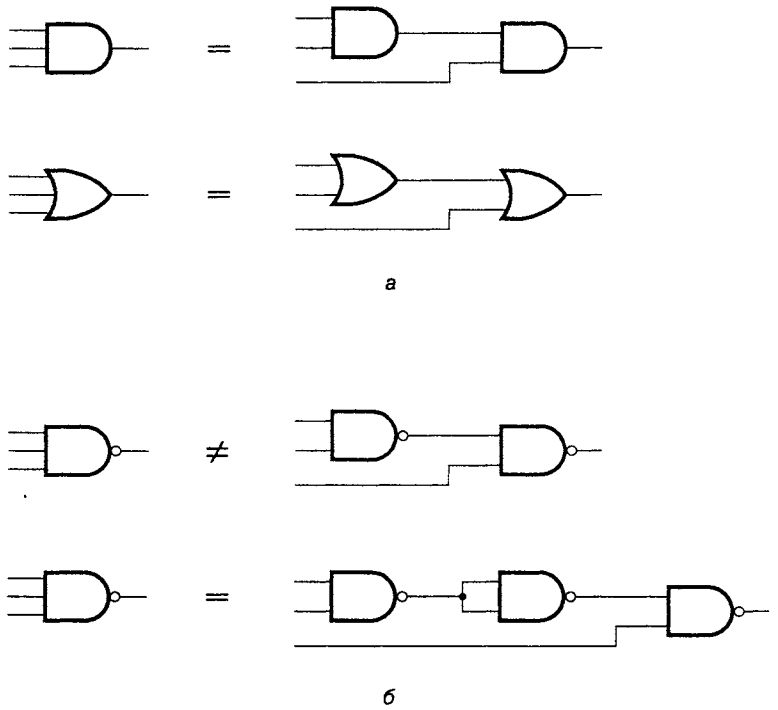


Рис. А.9. Реализация логических функций с тремя входными переменными на основе вентилей с двумя входами: И и ИЛИ (*а*); И-НЕ (*б*)

Аналогичным образом строятся схемы на основе вентилей ИЛИ-НЕ: любая логическая функция может быть представлена в виде суммы произведений и реализована в виде эквивалентной схемы на основе вентилей ИЛИ-НЕ.

Итак, вы познакомились с некоторыми базовыми концепциями проектирования логических схем.

Еще раз хотим обратить внимание читателя на тот факт, что для каждой логической функции существует множество вариантов реализации. Для практического воплощения функции важно выбрать вариант с наименьшей стоимостью. Кроме того, часто бывает необходимым минимизировать задержку распространения сигнала в логической схеме. С идеей минимизации схем мы познакомили вас в предыдущих разделах, чтобы объяснить суть процесса логического синтеза и показать,

за счет чего может быть достигнуто снижение стоимости реальных логических схем. Для этой цели можно применить карты Карно, которые подскажут, какие алгебраические операции приведут к оптимальному решению. Оптимизацию схем не обязательно выполнять вручную. К услугам конструкторов имеется сложное и достаточно мощное программное обеспечение для *автоматизированного проектирования* (Computer-Aided Design, CAD). Пользуясь такой программой, конструктор задает исходную функцию, и программа сама генерирует наиболее эффективную и простую схему для ее реализации.

А.5. Практическая реализация логических вентилей

Теперь мы перейдем к вопросу о практических средствах, используемых для представления логических переменных и логических функций. Совершенно очевидно, что выбор физического параметра, представляющего логические переменные, зависит от используемой технологии. В электронных схемах для этой цели может использоваться либо напряжение, либо сила электрического тока.

Для того чтобы установить соответствие между величиной напряжения и логическими значениями, или состояниями, в электронике используют концепцию *порога* (threshold). Напряжение, превышающее заданный порог, представляет одно значение, а напряжение, которое ниже этого порога, — совсем другое значение. На практике напряжение в любой точке электронной схемы подвержено небольшим случайным колебаниям, зависящим от множества причин. Из-за этого «шума» значения напряжения вблизи порога нельзя с уверенностью соотнести с конкретными логическими состояниями. Поэтому для электронных схем обычно устанавливают некоторый «запрещенный диапазон», как показано на рис. А.10. На этом рисунке напряжения ниже $V_{0,max}$ представляют логическое значение 0, а напряжения выше $V_{1,min}$ — значение 1. Далее в настоящей книге, говоря о напряжении, соответствующем логическим значениям 0 и 1, мы будем использовать понятия «низкое» и «высокое».

Мы начнем знакомство с электронными схемами, реализующими базовые логические функции, с описания простейших из них, которые состоят из резисторов и транзисторов, действующих в качестве переключателей. Для начала рассмотрим схемы, приведенные на рис. А.11. Когда ключ S на рис. А.11, *а* разомкнут, выходное напряжение V_{out} равно 0 («земля»). Когда же ключ S замкнут, выходное напряжение V_{out} равно напряжению источника V_{supply}. Точно так же действует схема, приведенная на рис. А.11, *б*, где роль ключа играет транзистор T. Когда входное напряжение, подаваемое на затвор транзистора, равно 0 (то есть когда $V_{in} = 0$), ключ разомкнут и $V_{out} = V_{supply}$. Когда же значение V_{in} изменяется на V_{supply}, ключ замыкается и выходное напряжение V_{out} становится близким к нулю. Таким образом, электронный ключ, схема которого показана на рис. А.11, *б*, может выполнять функции логического вентиля НЕ.

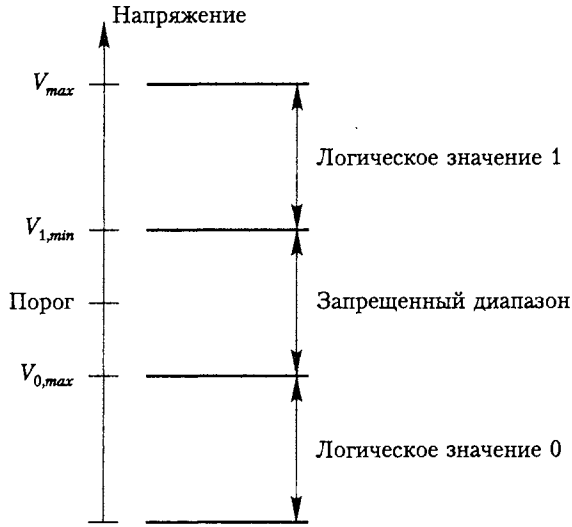


Рис. А.10. Представление логических значений посредством уровней напряжения

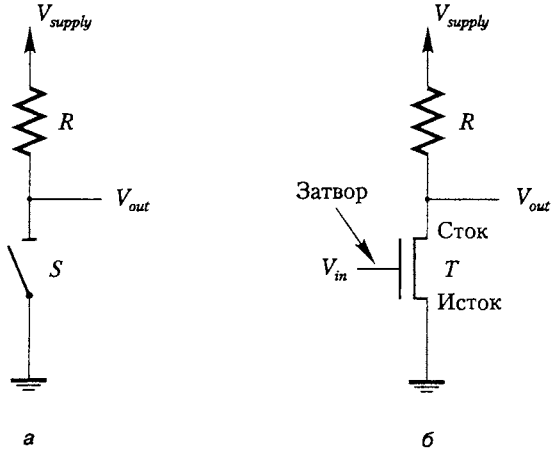


Рис. А.11. Схемы инвертора: с использованием ключа (*а*); с транзистором в качестве ключа (*б*)

На рис. А.12 показана электрическая цепь и эквивалентная ей электронная схема, реализующая вентиль ИЛИ-НЕ. На рис. А.12, *а* выходное напряжение V_{out} будет высоким только в том случае, если оба ключа, S_a и S_b, будут разомкнуты. Аналогичным образом, выходное напряжение V_{out} на рис. А.12, *б* будет высоким при условии, что входные напряжения V_a и V_b низки. Поэтому данная схема эквивалентна вентилю ИЛИ-НЕ, где входные напряжения V_a и V_b представляют две входные логические переменные, x_1 и x_2. Вентиль ИЛИ-НЕ можно сконструировать из двух транзисторов, соединив их последовательно (рис. А.13). Что касается логических функций И и ИЛИ, то они легко реализуются на основе вентилей

И-НЕ и ИЛИ-НЕ — достаточно вслед за вентилем включить в цепь инвертор, показанный на рис. А.11. Реализовать вентили И-НЕ и ИЛИ-НЕ несколько проще, чем И и ИЛИ. Поэтому неудивительно, что ими так часто пользуются при реализации логических функций. Стараясь сделать примеры как можно понятнее, мы приводили много схем на основе вентилей И, ИЛИ и НЕ. Но на практике логические схемы могут содержать вентили всех пяти типов.

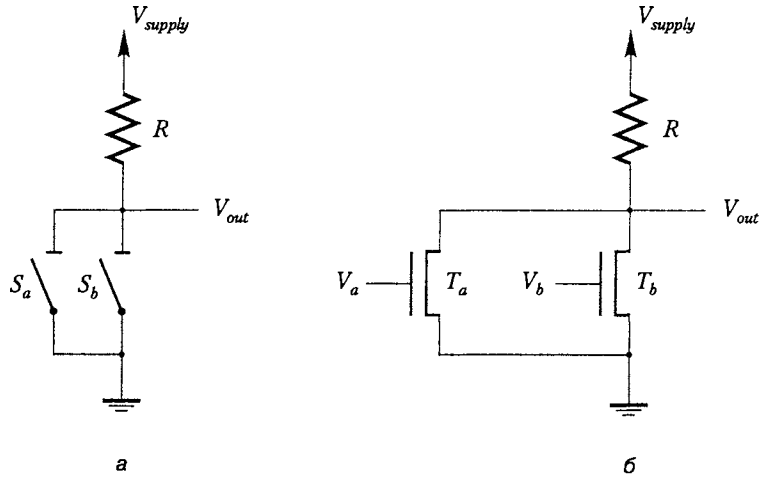


Рис. А.12. Схемы, реализующие вентиль ИЛИ-НЕ: с использованием двух ключей (*а*); с транзистором в качестве ключа (*б*)

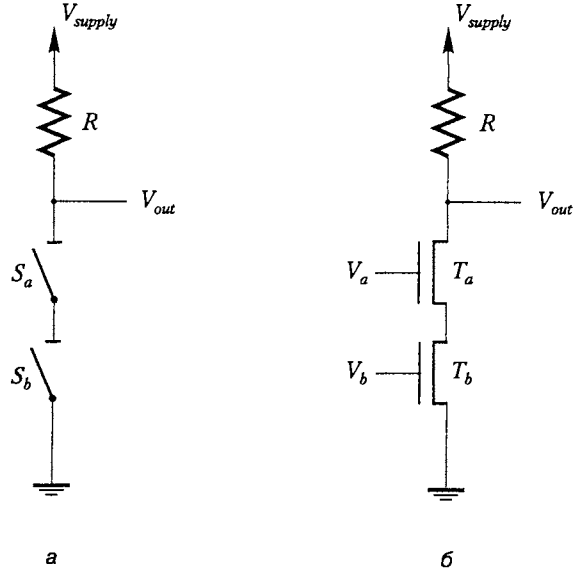


Рис. А.13. Схемы, реализующие вентиль И-НЕ: с использованием двух ключей (*а*); с транзистором в качестве ключа (*б*)

А.5.1. Схемы КМОП

Приведенные выше рис. А.11–А.13 отражают общую структуру электронных схем, создаваемых по *технологии n-МОП* (NMOS). В качестве ключей в электронных логических схемах используются *металло-оксидные полупроводниковые транзисторы* (МОП-транзисторы), которые бывают двух типов: n-канальные и p-канальные. N-канальные транзисторы называют транзисторами типа n-МОП. Когда на вход n-канального транзистора (то есть на его затвор) подается положительное напряжение источника питания, V_{supply}, ключ замыкается (рис. А.14, *а*). P-канальный транзистор действует наоборот: когда входное напряжение на его затворе V_G равно V_{supply}, ключ разомкнут, а когда $V_G = 0$, ключ замкнут (рис. А.14, *б*).

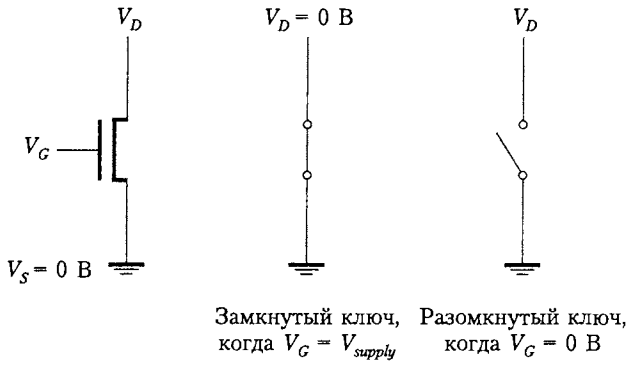


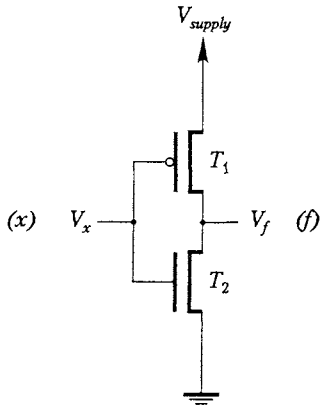
Рис. А.14. Логические схемы транзисторов: n-МОП (*а*); p-МОП (*б*)

Обратите внимание на графическое обозначение транзистора p-МОП: кружок на входе показывает, что его действие противоположно действию транзистора

n-МОП. Обратите также внимание, что у транзистора n-МОП символы s и d, обозначающие исток и сток, располагаются с противоположных сторон по сравнению с транзистором p-МОП. Исток n-канального транзистора соединяется с «землей», а исток p-канального транзистора — с источником напряжения V_{supply}. (Указанные обозначения отражают направление движения тока в транзисторах.)

Электронные схемы, приведенные на рис. А.11–А.13, имеют один недостаток: они требуют слишком большой мощности. В состоянии, когда ключи замкнуты и соединяют нагрузочный резистор R с «землей», электрический ток идет от источника напряжения V_{supply} к «земле». В противоположном состоянии, когда ключ разомкнут, соединения с «землей» нет, а значит, нет и тока. (В МОП-транзисторах ток через затвор не идет.) Таким образом, мощность, потребляемая электронными логическими схемами на МОП-транзисторах, зависит от состояния вентилей.

У данной проблемы имеется весьма эффективное решение: использовать в одной схеме транзисторы обоих типов, чтобы в устойчивом состоянии не потреблялась лишняя мощность. Эта идея была положена в основу технологии КМОП — построения схем на основе комплементарных (то есть дополняющих друг друга) металло-оксидных полупроводниковых транзисторов (Complementary Metal-Oxide Semiconductor, CMOS,). Суть КМОП-технологии иллюстрирует схема инвертора, приведенная на рис. А.15. Когда $V_x = V_{supply}$, что соответствует значению 1 входной переменой x, транзистор T_1 открыт, а транзистор T_2 закрыт. При этом выходное напряжение транзистора T_2 уменьшается от V_f до 0. Когда входное напряжение V_x становится равным 0, транзистор T_1 закрывается, а транзистор T_2 открывается. Выходное напряжение транзистора T_1 увеличивается до V_{supply}. Таким образом, логические значения x и f дополняют друг друга и схема реализует вентиль НЕ.



x	V_x	T_1	T_2	V_f	f
0	Низкое	Закрыт	Открыт	Высокое	1
1	Высокое	Открыт	Закрыт	Низкое	0

а *б*

Рис. А.15. КМОП-реализация вентиля НЕ: схема вентиля (*а*); таблица истинности и состояния транзисторов (*б*)

Суть этой схемы заключается в том, что транзисторы T_1 и T_2 действуют как логические дополнения: когда один из них закрыт, другой, наоборот, открыт. Поэто-

му точка выхода f всегда соединена либо с точкой V_{supply}, либо с «землей». При этом между «землей» и точкой V_{supply} никогда не бывает соединения, за исключением краткого переходного момента, когда изменяется состояние транзисторов. Это означает, что в устойчивом состоянии данная схема потребляет минимум энергии, и утечка происходит только в моменты перехода из одного логического состояния в другое. Таким образом, количество энергии, потребляемой логической схемой, зависит от частоты изменения состояния ее элементов.

Теперь концепцию КМОП можно распространить на схемы с n входами, как показано на рис. А.16. Транзисторы n-МОП используются для создания понижающей цепи, образующей соединение между точкой выхода f и «землей», когда реализуемая функция $F(x_1, ..., x_n)$ равна 0. Повышающая цепь создается на основе транзисторов p-МОП — она образует соединение между точкой выхода f и точкой V_{supply}, когда $F(x_1, ..., x_n) = 1$. Повышающая и понижающая цепи функционально дополняют друг друга, для того чтобы в устойчивом состоянии напряжение в точке f равнялось либо V_{supply}, либо нулю («земля»).

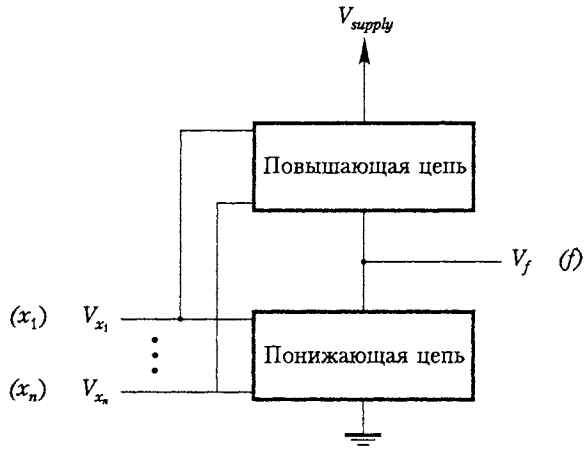
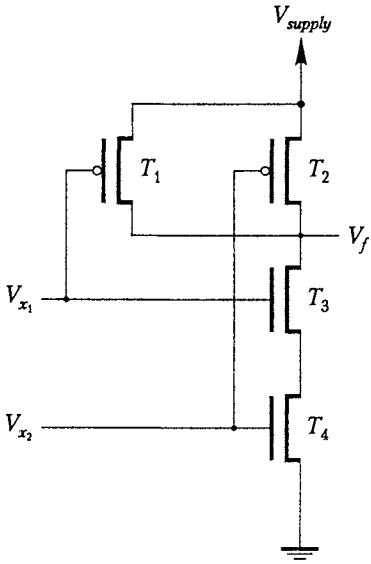


Рис. А.16. Структура КМОП-схемы

Понижающая цепь, подобно схемам, приведенным на рис. А.11–А.13, создается на основе транзисторов n-МОП. Как реализуются вентили И-НЕ и ИЛИ-НЕ, показано соответственно на рис. А.17 и А.18, а вентиль И, согласно рис. А.19, реализуется путем инвертирования выходного сигнала вентиля И-НЕ.

Значительное уменьшение потребляемой схемой мощности — это не единственное достоинство технологии КМОП. Еще одним ее преимуществом является очень маленький размер МОП-транзисторов. А это важно по двум причинам. Во-первых, на основе этих транзисторов производятся микросхемы с невероятно высокой степенью интеграции элементов: на одном современном чипе умещаются миллионы транзисторов, благодаря чему один чип может содержать мощный микропроцессор или блок памяти большой емкости. Во-вторых, чем меньше транзистор, тем быстрее он переключается из одного состояния в другое. Быстродействие современных интегральных КМОП-микросхем измеряется в гигагерцах.

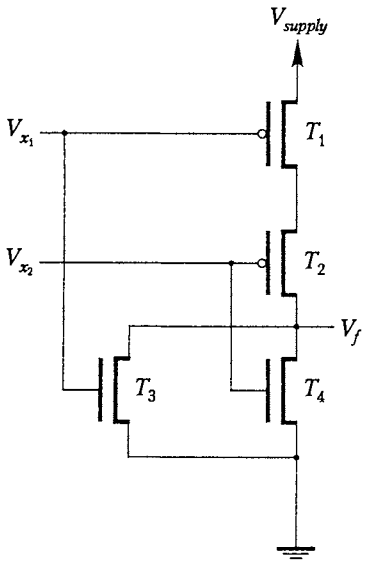


а

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	Закрыт	Закрыт	Открыт	Открыт	1
0	1	Закрыт	Открыт	Открыт	Закрыт	1
1	0	Открыт	Закрыт	Закрыт	Открыт	1
1	1	Открыт	Открыт	Закрыт	Закрыт	0

б

Рис. А.17. КМОП-реализация вентиля И-НЕ: схема вентиля (*а*); таблица истинности и состояния транзисторов (*б*)



а

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	Закрыт	Закрыт	Открыт	Открыт	1
0	1	Закрыт	Открыт	Открыт	Закрыт	0
1	0	Открыт	Закрыт	Закрыт	Открыт	0
1	1	Открыт	Открыт	Закрыт	Закрыт	0

б

Рис. А.18. КМОП-реализация вентиля ИЛИ-НЕ: схема вентиля (*а*); таблица истинности и состояния транзисторов (*б*)

Для современных КМОП-микросхем используются источники питания с напряжением от 1,5 до 15 В. Напряжение питания для наиболее распространенных микросхем равно 5 или 3,3 В. Чем меньше напряжение питания микросхемы, тем меньше потребляемая ею мощность (потребляемая мощность пропорциональна V_{supply}^2), а значит, на микросхему можно поместить большее количество транзисторов, не вызывая ее перегрева. К сожалению, снижение напряжения ведет к понижению помехоустойчивости микросхемы, так что здесь требуется разумный компромисс.

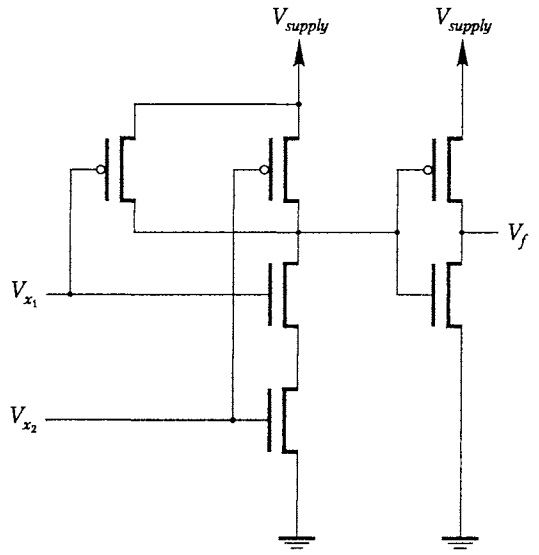


Рис. А.19. КМОП-реализация вентиля И

На рис. А.20 показано, как осуществляется переход между низким и высоким уровнями сигнала в КМОП-инверторе. Кривая, называемая *передаточной характеристикой*, отражает выходное напряжение как функцию входного напряжения. На данном рисунке видно, что когда входное напряжение проходит значение $V_{supply}/2$, выходное напряжение резко падает. Это значение входного напряжения, обозначенное на рисунке как V_t, называется *пороговым*. Отмеченное на графике значение δ, определяющее окрестность порогового напряжения, таково, что $V_{out} \approx V_{supply}$, если $V_{in} < V_t - \delta$, и $V_{out} \approx 0$, если $V_{in} > V_t + \delta$. Это значит, что для формирования правильного выходного сигнала входной сигнал не обязательно должен быть в точности равным номинальному значению 0 или V_{supply}. Допускается небольшая погрешность входного сигнала, называемая *шумом*, которая не вызывает нарушений в работе схемы. Приемлемые границы шума называются *запасом помехоустойчивости*. Для входного логического значения 1 запас помехоустойчивости составляет $V_{supply} - (V_t + \delta)$, а для значения 0 он равен $V_t - \delta$. У микросхем КМОП запас помехоустойчивости очень высок.

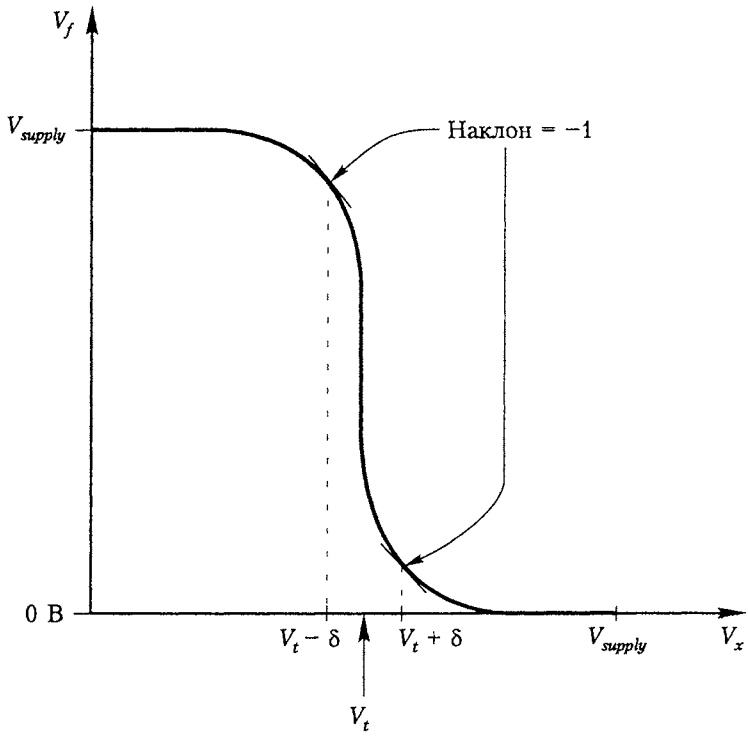


Рис. А.20. Передаточная характеристика КМОП-инвертора

А.5.2. Задержка на распространение сигнала

В электронных логических схемах переключение из одного состояния в другое происходит не мгновенно. Поэтому скорость работы схемы определяется тем, насколько быстро может измениться ее состояние. Соответствующий параметр схемы называется *задержкой на распространение сигнала*. Как он вычисляется, показано на рис. А.21. Выходное состояние изменяется с некоторой задержкой относительно момента изменения входного состояния. Обычно задержка определяется как интервал времени между точками, лежащими посередине графиков переходных характеристик входного и выходного сигналов (рис. А.21). Еще одной важной характеристикой схемы является *время перехода*, измеряемое как время между точками с 10- и 90-процентным изменением сигнала. С увеличением задержки на распространение сигнала по различным маршрутам максимальная скорость функционирования логической схемы снижается.

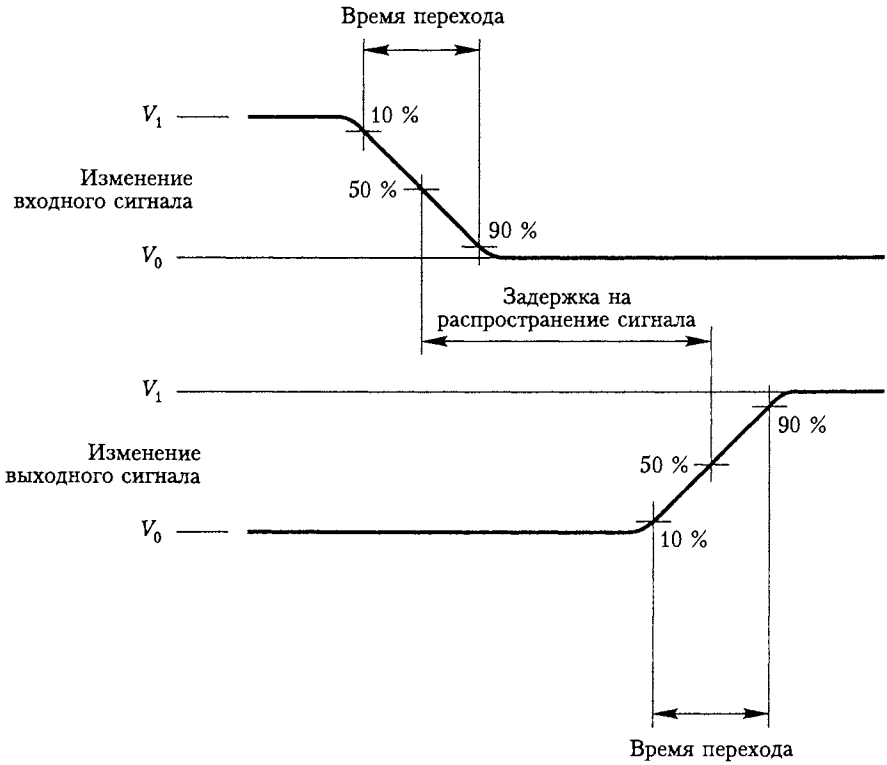


Рис. А.21. Задержка на распространение сигнала и время перехода

А.5.3. Ограничения по входу и выходу

Количество входов логического вентиля называется его *нагрузочной способностью по входу* или *коэффициентом объединения по входу* (fan-in). Количество входов логических вентилей, с которыми соединен выход данного вентиля, называется его *нагрузочной способностью по выходу* или *коэффициентом разветвления по выходу* (fan-out). В реальных микросхемах значения этих двух параметров невелики, поскольку с их увеличением возрастает задержка на распространение сигнала, а следовательно, снижается скорость работы схемы. Каждый транзистор повышает общее сопротивление вентиля в КМОП-схеме, а с увеличением сопротивления работа схемы замедляется и такие ее характеристики, как уровни сигнала и запас помехоустойчивости, ухудшаются. Поэтому нагрузочную способность по входу и по выходу обычно ограничивают до значений, не превышающих 10. Если исходная схема предполагает наличие вентиля с большим количеством входов, в нее просто добавляют еще один вентиль того же типа. Пример каскадирования однотипных вентилей вы видели на рис. А.9. Если же количество выходов вентиля превышает допустимый предел, можно просто использовать две копии этого вентиля.

А.5.4. Буферы с тремя состояниями

В логических вентилях, о которых шла речь до сих пор, выходы двух вентилей не могут соединяться между собой. С логической точки зрения, такое соединение просто не имеет смысла, поскольку в том случае, если один вентиль генерирует на выходе значение 1, а другой — значение 0, непонятно, что должен означать объединенный сигнал. Но что еще более важно, в микросхемах КМОП значение 1 на выходе вентиля соответствует созданию прямого соединения между выходом вентиля и точкой V_{supply}, тогда как выходное значение 0 соответствует соединению с землей. Поэтому, если соединить выходы двух вентилей в тот момент, когда на них будут противоположные значения, источник питания соединится с «землей» и произойдет короткое замыкание, которое может повредить вентили.

Конечно, при проектировании компьютерных систем нередко встречаются ситуации, когда входной сигнал схемы может исходить от одного из множества разных источников. В таких случаях используются мультиплексорные логические схемы, обсуждаемые в разделе А.10. В качестве альтернативы можно использовать специальные вентили, называемые *буферами с тремя состояниями*. У такого вентиля возможны не два, а три состояния. Он может генерировать обычные сигналы, то есть 0 и 1, а его третье состояние определяется состоянием выходного контакта, в котором он электрически отсоединен от входа того вентиля, которым должен управлять.

Принципиальная схема буфера с тремя состояниями показана на рис. А.22, *а*. У него два входа и один выход. Работой буфера управляет *разрешающий вход*, обозначенный на схеме как *e*. Когда $e = 1$, на выходе вентиля *f* то же логическое значение, что и на входе *x*. Когда $e = 0$, выход находится в высокоимпедансном состоянии *Z*. Эквивалентная логическая схема показана на рис. А.22, *б*. Треугольник на этом рисунке представляет неинвертирующий *повторитель*. Это схема, не выполняющая никакой логической операции, — на ее выходе просто повторяется входной сигнал. Если объединить ее с ключом, как на рис. А.22, *б*, новая схема будет работать в соответствии с таблицей истинности, приведенной на рис. А.22, *в*. Эта таблица определяет функцию с тремя состояниями. Как она реализуется, показано на рис. А.22, *г*. Два параллельно соединенных транзистора, n-МОП и p-МОП, образуют ключ, соединенный с выходом повторителя. Поскольку входные сигналы транзисторов этих двух типов противоположны, необходим инвертор. Когда $e = 0$, оба транзистора открываются, размыкая ключ. Когда $e = 1$, оба транзистора закрываются, замыкая ключ.

Схема повторителя может управлять множеством входов других вентилей, общее количество которых превышает возможности обычной логической схемы. Для этого в схеме повторителя нужны транзисторы большего размера. Потому-то в нее и включена цепь из двух вентилей НЕ, реализованных на основе более крупных транзисторов, чем те, которые обычно используются в логических вентилях.

Читатель, возможно, заинтересуется, для чего в выходном ключе понадобился транзистор p-МОП, если, с логической точки зрения, достаточно одного транзистора n-МОП. Дело в том, что эти транзисторы должны «передавать» логическое значение, генерируемое драйверной схемой, на выход *f*. Особенность описанных транзисторов заключается в том, что транзистор n-МОП хорошо передает логи-

ческое значение 0, но плохо передает значение 1, а транзистор p-МОП, напротив, хорошо передает логическое значение 1, но плохо передает значение 0. Если параллельно соединить эти два транзистора, оба значения будут передаваться одинаково хорошо.

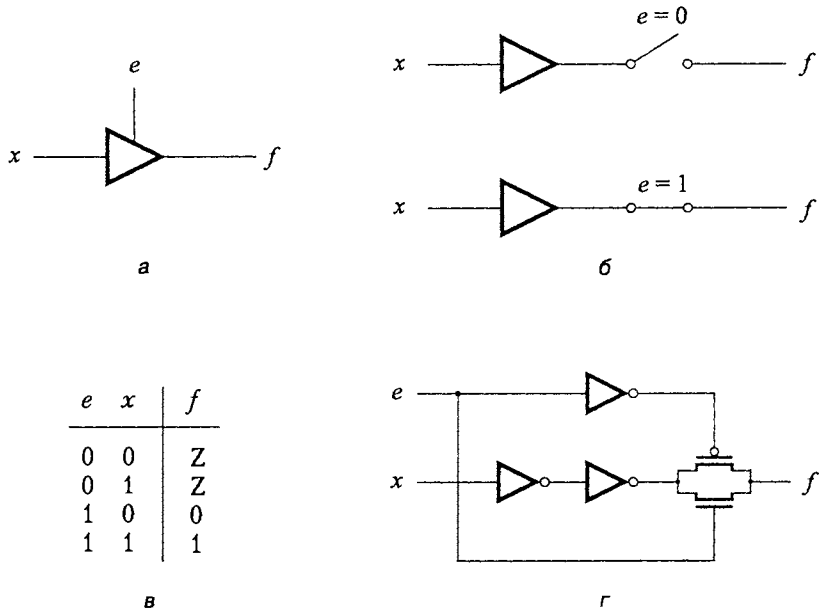


Рис. А.22. Буферы с тремя состояниями: символическое представление (*а*); эквивалентная схема (*б*); таблица истинности (*в*); реализация (*г*)

А.5.5. Модули интегральных микросхем

В предыдущих разделах были рассмотрены базовые принципы построения электронных схем для реализации логических функций. Для их практического воплощения используются серийно производимые интегральные схемы (ИС). Когда в 1960-х годах появились первые интегральные схемы, логические вентили стали выпускать в виде стандартизированных чипов. Такой чип монтировался в полностью закрытый защитный корпус со множеством металлических контактов для соединения с внешним устройством. Стандартные модули ИС имели разное количество контактов. На рис. А.23 показан простейший модуль с четырьмя вентилями И-НЕ. Эти четыре вентиля имеют общие контакты для соединения с «землей» и источником питания. Подобные ИС, содержащие всего несколько логических вентилей, называются *схемами с малой степенью интеграции* или *малыми интегральными схемами* (*МИС*).

Для столь простых функций, которые выполняют малые интегральные схемы, они занимают слишком много физического пространства. Более того, их производительность довольно низка из-за электрических характеристик контактов модуля ИС. Чтобы сгенерировать сигналы достаточной мощности, необходимые для

управления устройствами и схемами, подсоединенными к внешним контактам, приходится использовать большие транзисторы. В результате задержка на распространение сигнала и потребляемая схемой мощность заметно увеличиваются.

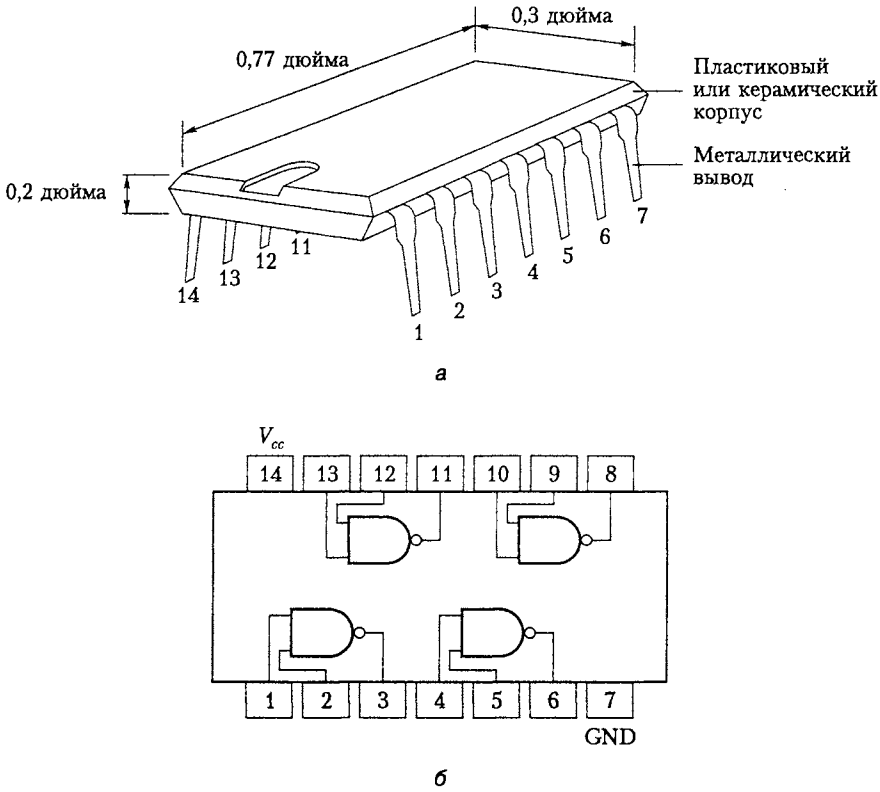


Рис. А.23. Модуль 14-контактной интегральной схемы: внешний вид (*а*); логическая структура схемы с четырьмя 2-входовыми вентилями И-НЕ (*б*)

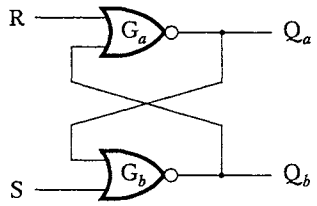
Время задержки на распространение сигнала для КМОП-вентиля И-НЕ (подобен показанному на рис. А.23), который входит в состав модуля интегральных схем, может достичь 5 нс. В случае таких же вентилей, используемых в больших интегральных схемах КМОП, задержка обычно составляет не более 0,2 нс, что зависит от технологии производства.

В настоящее время производятся гораздо более крупные ИС, реализующие разнообразные логические элементы. Чип интегральной схемы может содержать либо полезный функциональный блок, такой как сумматор, умножитель, регистр, шифратор или дешифратор, либо просто набор вентилей и программируемых переключателей внутренних соединений, с помощью которых конструктор может реализовать множество разных функций. В следующих разделах мы обсудим некоторые наиболее часто используемые функциональные блоки и программируемые пользователем логические устройства.

А.6. Триггеры

Большинству устройств, в которых задействована цифровая логика, требуются элементы для хранения информации. Например, схема управления кодовым замком должна запоминать последовательность открывающего его набора цифр. Еще один важный пример — электронная память для хранения данных, необходимая цифровым компьютерам. Базовый электронный элемент, используемый для хранения информации, называется *защелкой* (latch).

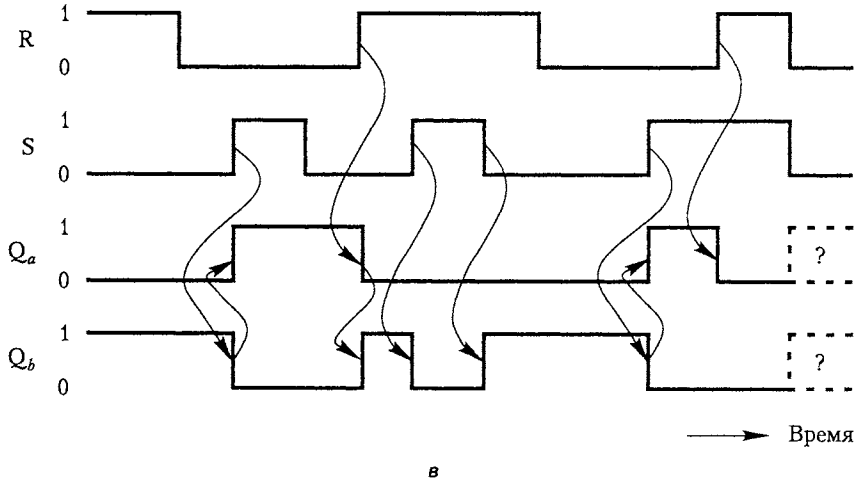
Давайте рассмотрим два вентиля ИЛИ-НЕ с перекрестным соединением, показанные на рис. А.24, *а*. Начнем с ситуации, когда R = 1, а S = 0. Очевидно, что в этом случае $Q_a = 0$, а $Q_b = 1$. Сигналы на обоих входах вентиля G_a равны 1. Поэтому, если сигнал на входе R изменить на 0, на выходах Q_a и Q_b ничего не изменится. Но если сигнал на входе R изменить на 0, а сигнал на входе S — на 1, то на выходах Q_a и Q_b появятся соответственно значения 1 и 0, причем они будут сохраняться даже после того, как сигнал на входе S снова станет равным 0. Описанная логическая схема представляет собой запоминающий элемент или защелку — она запоминает, на каком из двух входов (R или S) подавался последний сигнал 1.



а

S	R	Q_a	Q_b	
0	0	1/0	1/0	(без изменения)
0	1	0	1	
1	0	1	0	
1	1	0	0	

б



в

Рис. А.24. Защелка на основе вентилей ИЛИ-НЕ: логическая схема (*а*); таблица истинности (*б*); временная диаграмма (*в*)

Таблица истинности этой схемы приведена на рис. А.24, *б*. На рис. А.24, *в* представлены простейшие временные диаграммы прохождения сигналов через защелку. Стрелки показывают причинно-следственные отношения между сигналами. Обратите внимание, что когда значения на входах R и S одновременно изменяются с 1 на 0, результирующее состояние не определено. На практике защелка перейдет в одно из двух своих стабильных состояний, но в какое именно — предсказать невозможно. Комбинация входных значений R = S = 1 в подобных защелках обычно не используется.

В соответствии с принципом действия данной электронной схемы ее контакты S и R называют входами *установки* и *сброса* (*s*et и *r*eset соответственно, откуда и их обозначения). Поскольку значения R = S = 1 обычно не используются, выходы Q_a и Q_b обозначаются как Q и $\overline{Q}$. Однако обозначение $\overline{Q}$ является просто символом, указывающим на второй выход защелки, а вовсе не дополнением сигнала Q, поскольку для набора входных значений R = S = 1 на выходе получается $Q = \overline{Q} = 0$.

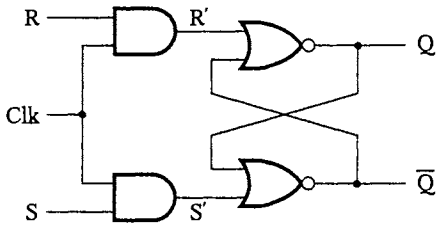
А.6.1. Вентильные защелки

На практике часто бывает необходимым, чтобы управление временем сброса и установки защелки осуществлялось посредством входного сигнала, отличного от сигналов R и S. Этот входной сигнал называется *синхронизирующим* или *тактовым* входом (clock input). Логический элемент с таким управлением получил название *вентильная SR-защелка*. Логическая схема, таблица истинности, временная диаграмма и графическое обозначение такой защелки продемонстрированы на рис. А.25. Когда сигнал на синхронизирующем входе Clk равен 1, сигналы в точках S′ и R′ равны входным сигналам S и R соответственно. Когда Clk = 0, сигналы в точках S′ и R′ тоже равны 0 и изменить состояние защелки невозможно.

До сих пор для описания поведения логических схем мы использовали таблицы истинности. В таблице истинности приводятся выходные значения схемы, соответствующие каждой комбинации входных значений. Те логические схемы, выходные значения которых уникально определены для каждого входного значения, называются *комбинаторными*. Именно схемы этого класса обсуждались в разделах А.1–А.4. Схемы, содержащие запоминающие элементы, относятся к другому классу и называются *последовательными* (рис. А.24). Выходное значение любой такой схемы является функцией не только текущих значений входных переменных, но и их предшествующего состояния.

Таблица истинности защелки нуждается в некоторой модификации, поскольку должна отражать предыдущее состояние схемы. Обратимся к приведенной на рис. А.25, *б* таблице истинности, которая описывает поведение вентильной SR-защелки. В ней предыдущее состояние схемы обозначено как $Q(t)$. Переход в следующее состояние, обозначаемое как $Q(t + 1)$, происходит после поступления тактового импульса. Обратите внимание, что для набора входных значений S = R = 1 значение $Q(t + 1)$ не определено по причинам, о которых рассказывалось выше.

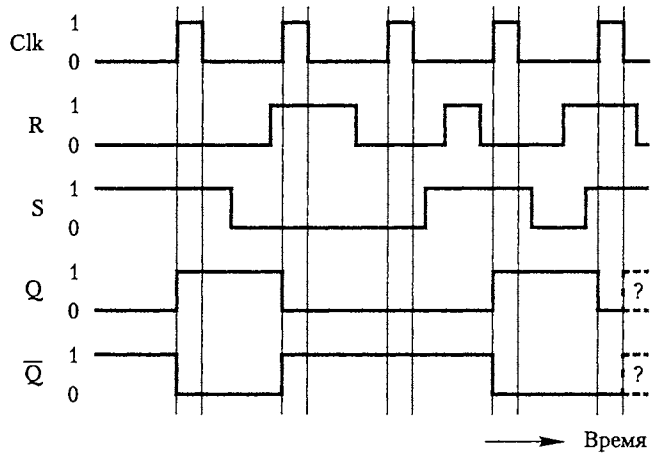
Как следует из рис. А.26, вентильная SR-защелка может быть реализована с помощью вентилей И-НЕ. Рекомендуем вам выполнить полезное упражнение и доказать, что данная схема функционально эквивалентна схеме, приведенной на рис. А.25, *а* (см. упражнение А.20).



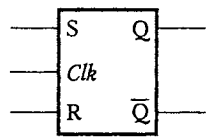
а

Clk	S	R	Q(*t* + 1)
0	x	x	Q(*t*) (без изменения)
1	0	0	Q(*t*) (без изменения)
1	0	1	0
1	1	0	1
1	1	1	x

б



в



г

Рис. А.25. Вентильная SR-защелка: схема (*а*); таблица истинности (*б*); временная диаграмма (*в*); графическое обозначение (*г*)

На рис. А.27 показан еще один вид вентильной защелки, называемый *вентильной D-защелкой*. В этой схеме сигналы S и R поступают из одного входа, обозначенного как D. В ответ на поступление тактового импульса значение на выходе Q становится равным 1, если D = 1, или сбрасывается в 0, если D = 0. Это означает, что при поступлении каждого нового тактового импульса сигнал на входе D передается на выход Q D-защелки и сохраняется неизменным до прихода следующего тактового импульса.

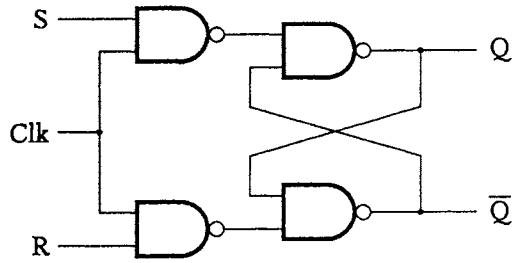
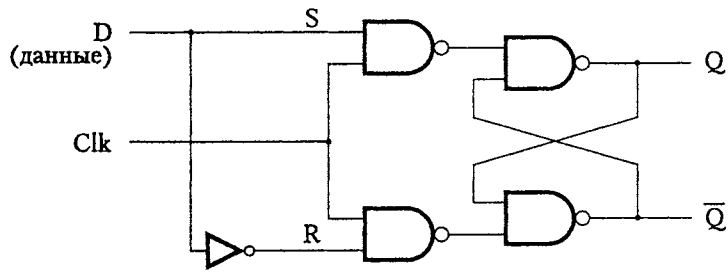


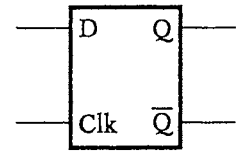
Рис. А.26. Вентильная SR-защелка, реализованная на основе вентилей И-НЕ



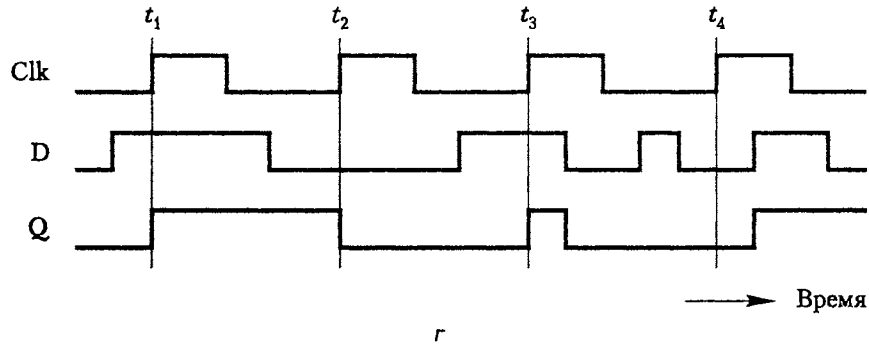
а

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

б



в



г

Рис. А.27. Вентильная D-защелка: схема (*а*); таблица истинности (*б*); графическое обозначение (*в*); временная диаграмма (*г*)

А.6.2. Двухступенчатые триггеры

В схеме, представленной на рис. А.25, предполагалось, что пока Clk = 1, значения на входах S и R не изменяются. Однако, проанализировав эту схему, вы увидите, что ее выходы немедленно реагируют на любые изменения значений на входах S и R. Аналогичным образом предполагается, что в схеме, приведенной на рис. А.27, Q = D, пока Clk = 1. Однако во многих случаях синхронное изменение сигнала на выходе, немедленно отражающее изменение сигнала на входе, недопустимо. В частности, описанные выше защелки типа D и SR не годятся для создания счетчиков и сдвиговых регистров, о которых мы поговорим позднее. В таких схемах мгновенная передача логического условия от входов (R, S и D) к выходам защелки может привести к ее неверному функционированию. Эту проблему призвана решать концепция *ведущий-ведомый*. Две вентильные D-защелки могут быть объединены в схему (рис. А.28, *а*), которая получила название *синхронный двухступенчатый триггер* или *D-триггер «ведущий-ведомый»*. Первая защелка, называемая ведущей, соединена со входом D второй защелки, пока сигнал Clock равен 1. Изменение значения на входе Clock с 1 на 0 изолирует ведущую защелку от входа D и передает ее содержимое в ведомую. Но ни при каком состоянии защелок прямого пути от входа D до выхода Q не существует.

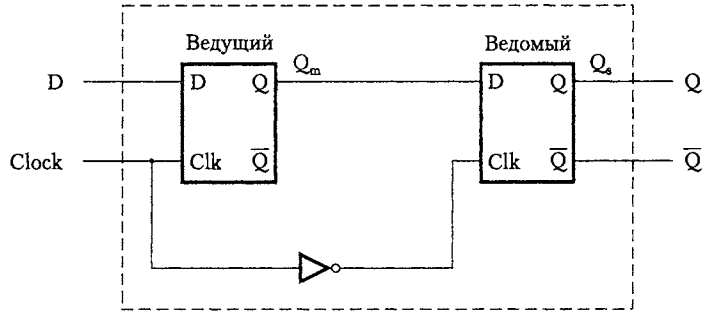
Следует заметить, что пока Clock = 1, изменения на входе триггера D немедленно отражаются на состоянии ведущей защелки. Предназначение ведомой защелки заключается в том, чтобы сохранять значение на выходе триггера, несмотря на переход ведущей защелки в следующее состояние, определяемое входом D. Это новое состояние передается ведомой защелке в тот момент, когда значение на входе Clock изменяется с 1 на 0. С этого момента ведущая защелка изолируется от входов ведомой, а следовательно, дальнейшие изменения на входе D не будут отражаться на состоянии ведомой защелки. Примеры изменения состояний входов и выходов такого D-триггера приведены на рис. А.28, *б*.

Термином *триггер* (flip-flop) называют запоминающий элемент, выходное состояние которого меняется на фронте управляющего тактового сигнала (то есть в момент его перехода из одного состояния в другое). В описанном выше двухступенчатом D-триггере видимые изменения происходят на отрицательном фронте сигнала (то есть в момент его перехода из 1 в 0). Изменение становится видимым, когда оно достигает терминала Q ведомой защелки. Обратите внимание, что в схеме на рис. А.28 ведущей защелкой можно управлять с помощью дополнения сигнала Clock, а ведомой — с помощью исходного сигнала Clock. В последнем случае изменения на выходе триггера Q будут происходить на положительном фронте тактового сигнала.

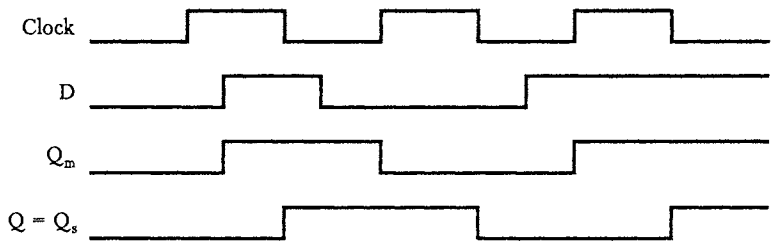
Графическое изображение триггера показано на рис. А.28, *в*. Его тактовый вход обозначается стрелкой (а не с помощью пометки Clk). Это стандартное обозначение, указывающее, что изменение состояния триггера происходит на положительном фронте тактового сигнала. Поскольку на нашем рисунке представлена ситуация, когда изменение происходит на отрицательном фронте, на тактовом входе перед стрелкой отображается маленький кружок.

А.6.3. Тактирование фронтом сигнала

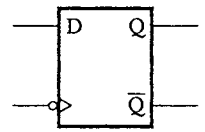
Триггер называется *тактируемым фронтом сигнала*, если поданные на его вход данные передаются на выход только в момент изменения тактового сигнала. Все остальное время вход и выход изолированы друг от друга. Термины *тактируемый положительным (передним) фронтом сигнала* и *тактируемый отрицательным (задним) фронтом сигнала* относятся к триггерам, в которых передача данных происходит в ответ на изменение тактового сигнала соответственно с 0 на 1 и с 1 на 0. Для корректного функционирования триггера, тактируемого фронтом сигнала, необходимо, чтобы фронт тактового сигнала был четко определен и имел очень малое время перехода. На рис. А.28 изображен двухступенчатый триггер, тактируемый отрицательным фронтом сигнала.



а



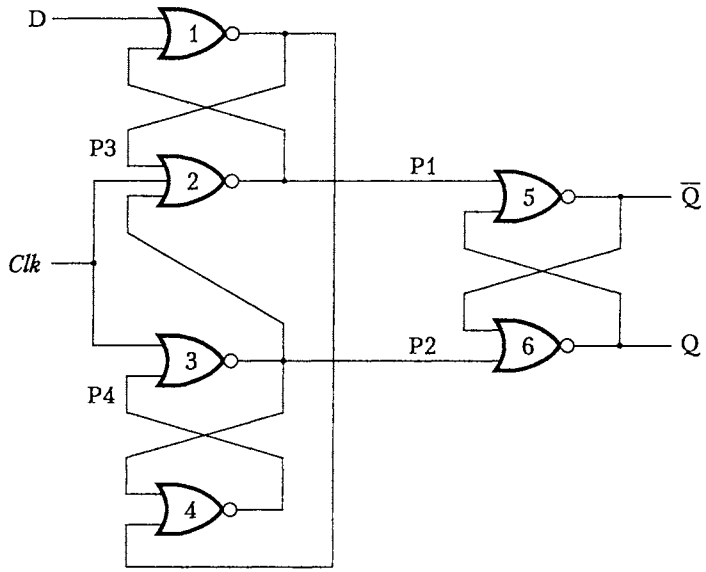
б



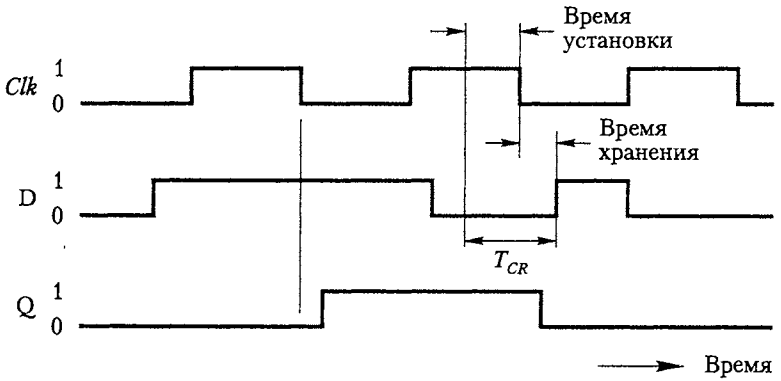
в

Рис. А.28. Двухступенчатый D-триггер: схема (*а*); временная диаграмма (*б*); графическое обозначение (*в*)

Другой способ реализации двухступенчатого триггера, тактируемого отрицательным фронтом сигнала, показан на рис. А.29, *а*.



а



б

Рис. А.29. D-триггер, тактируемый отрицательным фронтом сигнала: схема (*а*); пример временной диаграммы (*б*)

Посмотрим, как действует этот триггер. Если Clk = 1, на выходы вентилей 2 и 3 подается сигнал 0. Поэтому состояние на выходах триггера Q и $\overline{Q}$ сохраняется неизменным. Нетрудно убедиться, что пока Clk = 1, точки P3 и P4 немедленно отражают изменения на входе D. Сигнал в точке P3 остается равным сигналу $\overline{D}$, а сигнал

в точке P4 — равным D. Когда на вход Clk поступает сигнал 0, эти значения передаются в точки P1 и P2 с помощью вентилей 2 и 3 соответственно. Таким образом выходная защелка, состоящая из вентилей 5 и 6, переходит в новое состояние, которое она теперь должна хранить.

Мы сможем убедиться в том, что дальнейшие изменения значений на входе D при Clk = 0 не влияют на сигналы в точках P1 и P2, рассмотрев два случая. Для начала предположим, что на отрицательном фронте сигнала Clk вход D равен 0. Единичный сигнал в точке P2 сохраняет значение 1 на соответствующих входах вентилей 2 и 4, в результате чего, независимо от дальнейших изменений сигнала на входе D, в точках P1 и P2 сохраняются соответственно значения 0 и 1. Теперь предположим, что на отрицательном фронте сигнала Clk вход D = 1. Сигнал 1 в точке P1 означает, что дальнейшие изменения на входе D не могут воздействовать на выход вентиля 1, где сохраняется сигнал 0.

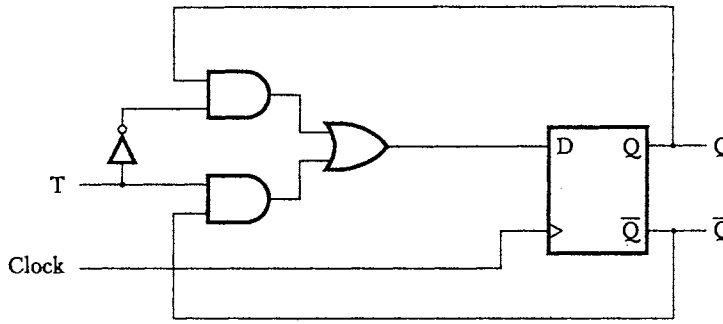
Когда в начале следующего тактового импульса значение Clk меняется на 1, в точках P1 и P2 снова появляется сигнал 0, изолирующий выход от остальной части схемы. После этого точки P3 и P4 отражают, как было сказано выше, изменения в точке D.

Как работает этот тип D-триггера, показано на рис. А.29, *б*. Значение на выходе, которое появится после того, как сигнал на входе Clk изменится с 1 на 0, будет равно значению на входе D триггера непосредственно перед этим переходом. Однако непосредственно перед отрицательным фронтом сигнала Clk и сразу после такового существует критический период времени T_{CR}, в течение которого значение D не должно изменяться. Этот промежуток времени, как показано на рисунке, разделяется на две части: время установки и время хранения. На временной диаграмме видно, что выходное значение Q изменяется с небольшой задержкой после отрицательного фронта тактового сигнала. Причиной этого является задержка на распространение сигнала в вентилях ИЛИ-НЕ.

А.6.4. Т-триггеры

Наиболее часто используемым типом триггеров являются D-триггеры, поскольку они могут использоваться для временного хранения данных. Однако во многих случаях требуются триггеры других типов. Например, схемы счетчиков, о которых упоминалось в разделе А.8, эффективнее реализуются на основе триггеров типа Т. Состояние *Т-триггера* изменяется на каждом такте, если на его вход Т подается значение 1. Говорят, что такой триггер «переключает» свое состояние.

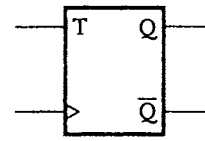
Схема Т-триггера, а также его таблица истинности, графическое обозначение, пример временной диаграммы представлены на рис. А.30, *а*. Как следует из рисунка, в основе Т-триггера лежит D-триггер. Обратите внимание, что D-триггер тактируется положительным фронтом сигнала.



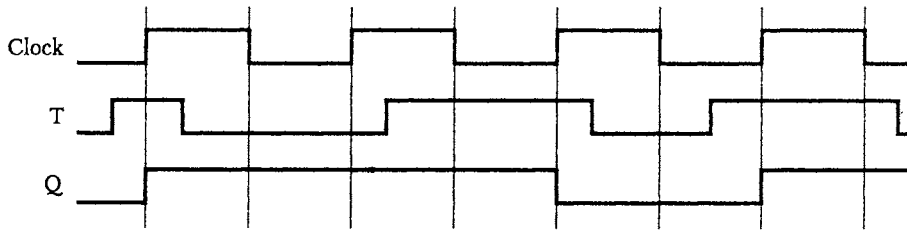
а

T	Q(t + 1)
0	Q(t)
1	$\overline{Q}$(t)

б



в



г

Рис. А.30. Т-триггер: схема (*а*); таблица истинности (*б*); графическое обозначение (*в*); временная диаграмма (*г*)

А.6.5. JK-триггеры

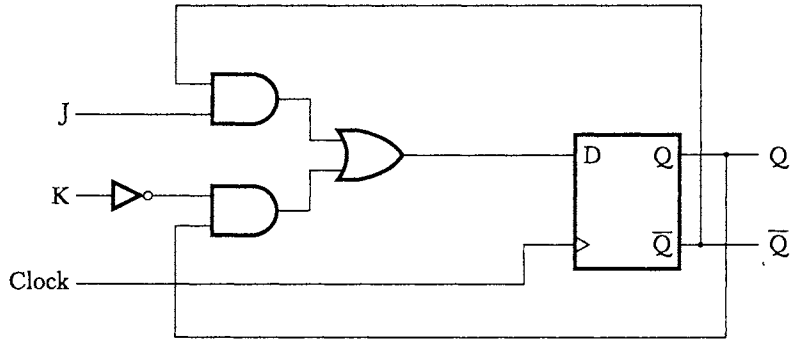
JK-триггеры — это еще один тип триггеров, который, правда, используется менее часто, нежели описанные выше. JK-триггер обладает чертами триггеров SR и T. Его схема, таблица истинности и обозначение приведены на рис. А.31. Первые три строки в таблице истинности JK-триггера определяют его поведение, аналогичное поведению вентильной SR-защелки при Clk = 1 (рис. А.25, *б*), так что входы J и K соответствуют входам S и R. При входном сигнале J = K = 1 следующее состояние триггера определяется как дополнение его текущего состояния. Это означает, что когда J = K = 1, триггер действует как переключатель, изменяя свое текущее состояние на противоположное.

JK-триггер можно реализовать на основе D-триггера и нескольких вентилей, соединенных таким образом, что

$$D = J\overline{Q} + \overline{K}Q$$

Соответствующая схема показана на рис. А.31, *а*.

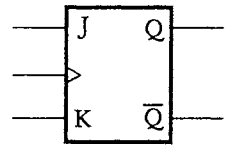
Триггеры JK могут применяться для разных целей. В частности, их подобно D-триггерам можно использовать для хранения данных. На их основе также можно создавать счетчики, поскольку при соединении терминалов J и K они ведут себя как T-триггеры.



а

J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\overline{Q}(t)$

б



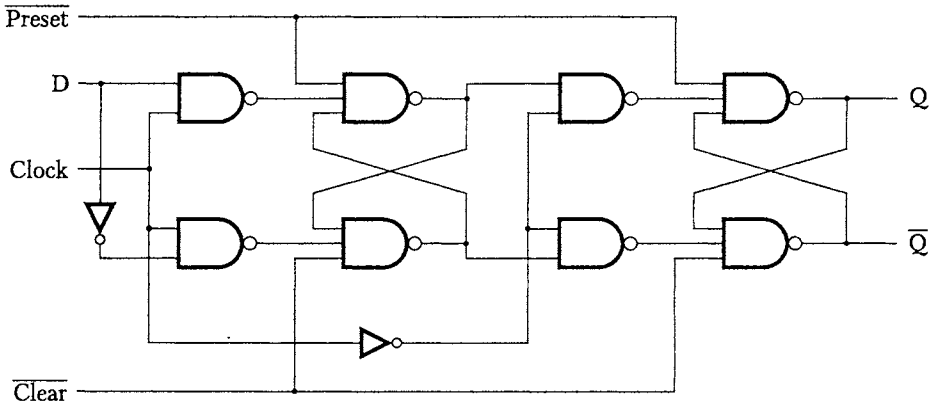
в

Рис. А.31. JK-триггер: схема (*а*); таблица истинности (*б*); графическое обозначение (*в*)

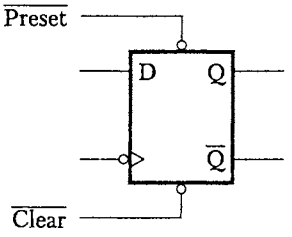
А.6.6. Триггеры с дополнительными входами для установки и очистки

Состояние триггера определяется его предшествующим состоянием и логическими значениями на входных терминалах. Иногда для триггера нужно задать конкретное состояние вне зависимости от его текущего состояния и значений на стандартных входах. Например, определенное состояние триггеров задается при включении компьютера. Обычно это означает, что значения на выходах всех триггеров устанавливаются в 0. Лишь в отдельных случаях они могут быть равными 1.

На рис. А.32 показано, как добавить в двухступенчатый D-триггер управляющие сигналы установки и очистки, которые переводят триггер в состояние 1 или 0 независимо от сигналов на входе D и на тактовом входе. Как видно из схемы, на указанные входы подается сигнал низкого уровня. Когда оба сигнала на входах $\overline{\text{Preset}}$ (установка) и $\overline{\text{Clear}}$ (очистка) равны 1, триггером, как обычно, управляют входы Clock и D. Когда на вход $\overline{\text{Preset}}$ подается сигнал 0, триггер переходит в состояние 1, а когда $\overline{\text{Clear}} = 0$, триггер устанавливается в состояние 0. Управляющие сигналы установки и очистки часто добавляются и в триггеры других типов.



а



б

Рис. А.32. Двухступенчатый D-триггер с входами установки и очистки: схема (*а*); графическое обозначение (*б*)

А.7. Регистры и сдвиговые регистры

Отдельный триггер может использоваться для хранения одного бита информации. Однако для машин, которые должны обрабатывать слова данных, состоящие из множества битов (обычно 64), удобнее объединить группу триггеров в стандартную структуру, называемую *регистром*. Работа триггеров, входящих в состав регистра, синхронизируется общим тактовым входом. Поэтому данные записываются (загружаются) во все триггеры и считываются изо всех триггеров одновременно. В ходе обработки цифровых данных часто требуется сдвинуть или циклически прокрутить значения группы битов данных. Реализуются эти операции аппаратно. Простейшим механизмом для их выполнения является регистр, содержимое которого легко может быть сдвинуто вправо или влево на одну позицию за раз. В качестве примера рассмотрим 4-разрядный сдвиговый регистр, показанный на рис. А.33. Он состоит из четырех D-триггеров, соединенных таким образом, что каждый тактовый импульс вызывает перемещение содержимого триггера F_i в триггер F_{i+1}, в результате чего получается «сдвиг вправо». Данные последовательно «вдвигаются» в регистр и «выдвигаются» из него. Для выполнения циклического смещения данных достаточно соединить выход Out и вход In.

Для корректного функционирования сдвигового регистра необходимо, чтобы на каждый тактовый импульс его содержимое смещалось ровно на одну позицию. Это условие накладывает некоторые ограничения на запоминающие элементы, которые могут использоваться в сдвиговых регистрах. Например, вентильные защелки (рис. А.27) для этой цели не подходят. При высоком уровне тактового сигнала значение на входе D немедленно передается на выход, а оттуда — на следующую защелку. В результате количество сдвигов на один тактовый импульс никак не контролируется. Поэтому сдвиговые регистры создаются на основе двухступенчатых триггеров или триггеров, тактируемых фронтом сигнала.

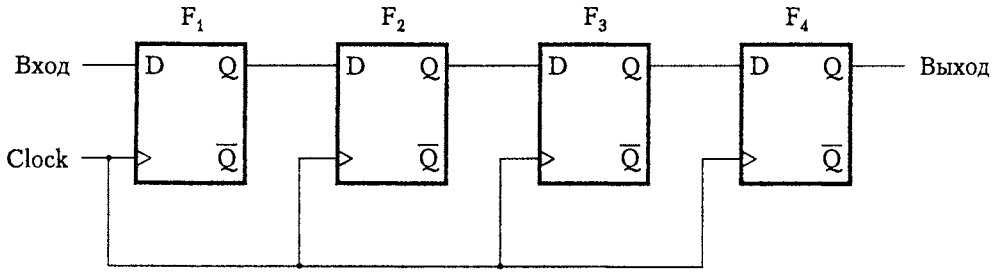


Рис. А.33. Простейший сдвиговый регистр

Интересной разновидностью сдвигового регистра является регистр, разряды которого могут считываться и загружаться параллельно. Для этого в него добавляются дополнительные вентильные схемы (рис. А.34). Загрузка данного регистра может выполняться как последовательно, так и параллельно. Когда на тактовый вход регистра подается очередной импульс, при условии, что $\overline{\text{Shift}}/\text{Load} = 0$, выполняется сдвиг, а в противном случае — параллельная загрузка регистра.

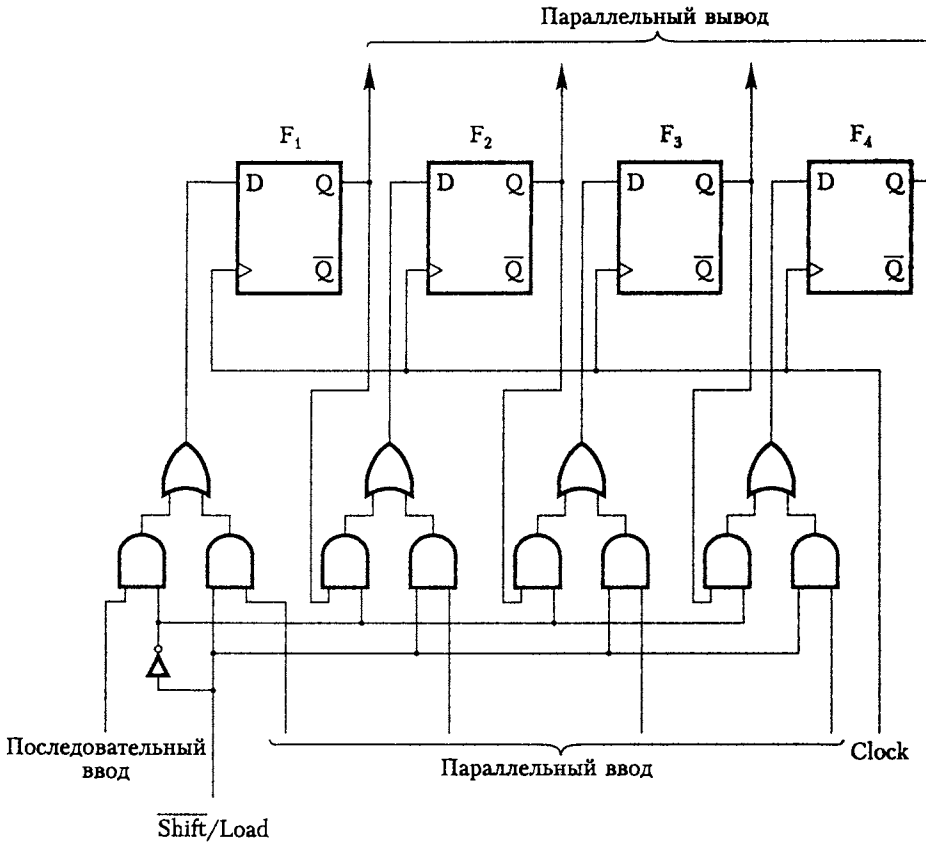


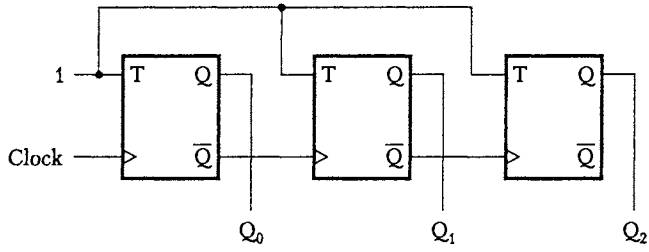
Рис. А.34. Сдвиговый регистр с параллельным доступом

А.8. Счетчики

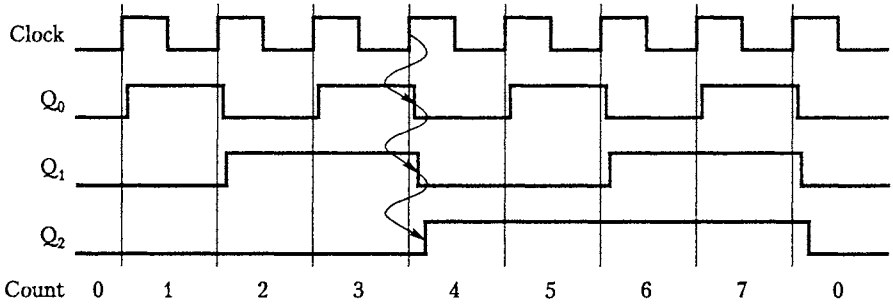
В предыдущем разделе рассказывалось о применении триггеров для создания сдвиговых регистров. Кроме того, триггеры используются в схемах *счетчиков*. Пожалуй, не нужно объяснять, для чего в цифровых компьютерах нужны счетчики. Но здесь речь идет не только об аппаратном механизме для выполнения обычных счетных функций — с их помощью можно также генерировать управляющие и тактирующие сигналы. Счетчик, управляемый высокочастотным тактовым сигналом, может использоваться для выдачи более редких сигналов кратной частоты. Такие счетчики называются *делителями частоты*.

Простейший трехступенчатый (или 3-разрядный) счетчик конструируется на основе Т-триггера (рис. А.35). Напомним, что когда на вход Т подается значение 1, триггер действует как переключатель, то есть его состояние изменяется при подаче каждого тактового импульса. Два последовательных тактовых импульса приводят к изменению выхода Q_0 — из состояния 1 в состояние 0 и опять в состояние 1 или же из состояния 0 в состояние 1 и опять в 0. Таким образом, частота

изменения выходного сигнала Q_0 будет вдвое меньшей, чем частота входного тактового сигнала. А в связи с тем, что второй триггер тактируется сигналом Q_0, частота изменения его выходного сигнала Q_1 будет вдвое меньшей, чем частота Q_0, и вчетверо меньшей, чем частота исходного тактового сигнала. В данном примере предполагается тактирование всех трех триггеров положительным фронтом.



а



б

Рис. А.35. 3-разрядный счетчик прямого счета: схема (*а*); временная диаграмма (*б*)

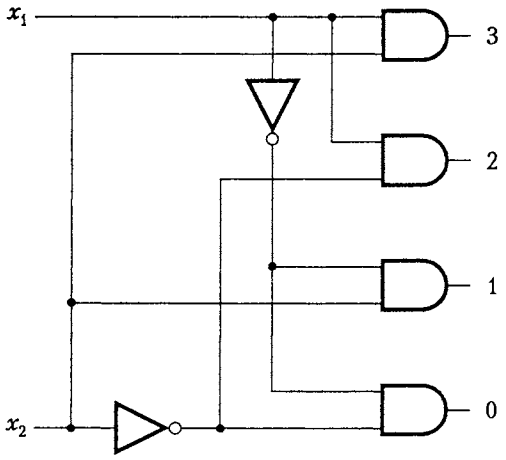
Такой счетчик называют *счетчиком со сквозным переносом* или *волнообразным счетчиком* (ripple counter), поскольку входной тактовый сигнал волнообразно распространяется по его схеме. Например, положительный фронт импульса 4 меняет сигнал 1 на выходе Q_0 на 0. Это изменение на выходе Q_0, в свою очередь, вызывает изменение сигнала на выходе Q_1 — из 1 в 0, что опять-таки, изменяет сигнал Q_2 — из 0 в 1. Если в каждом триггере происходит некоторая задержка Δ, то задержка перед установкой сигнала на выходе Q_2 составляет уже 3Δ. Если от счетчика требуется очень высокая скорость работы, подобная задержка может вызывать проблему. Однако время задержки по сравнению с тактовой частотой, как правило, очень мало, поэтому им можно пренебречь.

Добавив еще несколько логических вентилей, можно сконструировать *синхронный счетчик*, в котором все ступени будут управляться общим тактовым сигналом, так что состояния всех триггеров будут изменяться одновременно. Такие счетчики способны функционировать с очень высокой скоростью, поскольку общее время задержки на распространение сигнала в них существенно сокращено.

В противоположность им счетчики такой конструкции, как на рис. А.35, называются *асинхронными*.

А.9. Дешифраторы

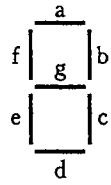
Значительная часть информации хранится и обрабатывается в компьютерах в закодированном виде. Например, если речь идет о машинной команде, то для ее хранения может использоваться n-битовое поле, вмещающее один из 2^n различных кодов операций. Но прежде чем выполнить требуемую операцию, закодированная команда должна быть декодирована. Схема, которая способна принять входное значение, состоящее из n разрядов, и сгенерировать соответствующий выходной сигнал на одной из 2^n выходных линий, называется *дешифратором* (или *декодером*). Простейший пример дешифратора с двумя входами и четырьмя выходами показан на рис. А.36. Одна из четырех выходных линий выбирается на основании значений на входах x_1 и x_2. На выбранный выход подается логическое значение 1, а на оставшиеся выходы — логическое значение 0.



x_1	x_2	Активный выход
0	0	0
0	1	1
1	0	2
1	1	3

Рис. А.36. Дешифратор с двумя входами и четырьмя выходами

Существуют и другие полезные типы дешифраторов. Так, при использовании двоично-десятичных данных обычно требуются декодирующие схемы, в которых четыре входные переменные, представляющие двоично-кодированное десятичное число, используются для выбора одного из 10 возможных выходов. В качестве еще одного специфического примера можно рассмотреть дешифратор, используемый для управления 7-сегментным индикатором. Структура соответствующего 7-сегментного элемента показана на рис. А.37. Как видите, с его помощью можно отобразить любую десятичную цифру. Соответствующие функции для каждого из 7 сегментов индикатора приведены в таблице истинности на рис. А.37. Они реализуются с помощью показанной на этом же рисунке электронной схемы, составленной из вентилей И-НЕ. Предлагаем читателю самостоятельно убедиться, что данная схема действительно реализует приведенные функции.



Цифра	x_1	x_2	x_3	x_4	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

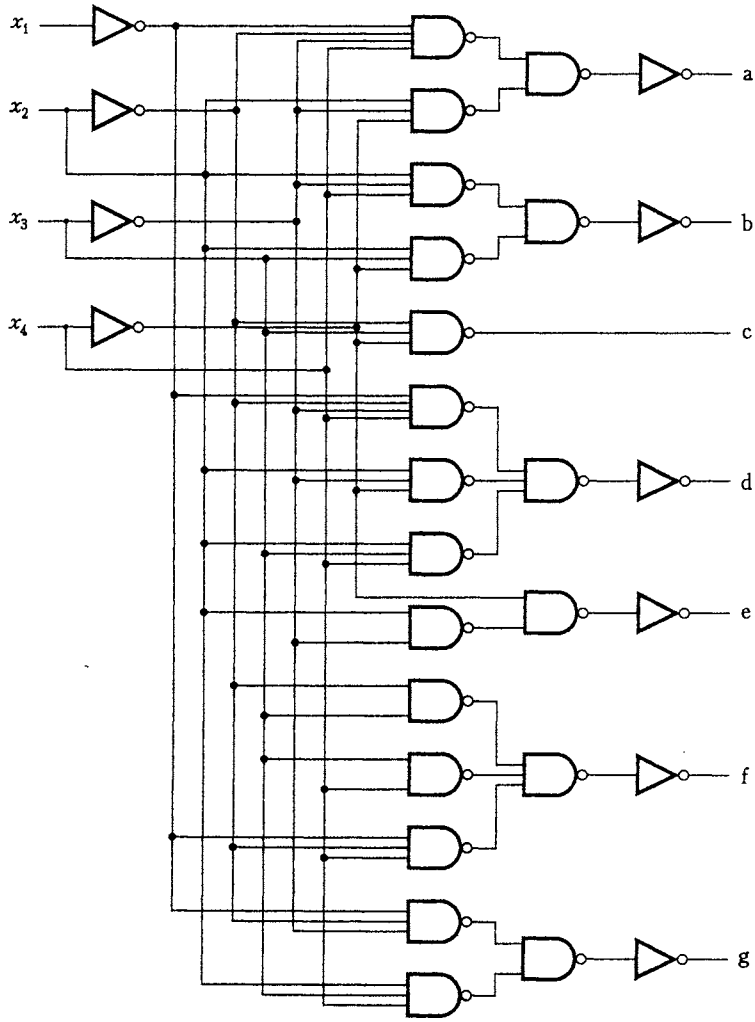
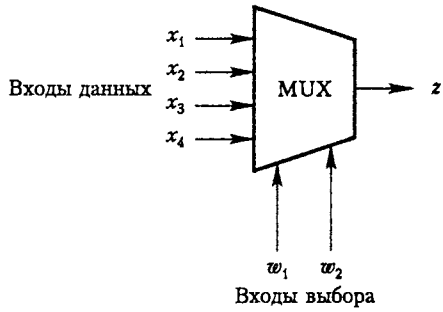


Рис. А.37. Дешифратор, преобразующий двоично-десятичное число для отображения на 7-сегментном индикаторе

А.10. Мультиплексоры

В предыдущем разделе рассказывалось о дешифраторах, устанавливающих на основе входных сигналов значение 1 на одной из выходных линий. На выбранную линию дешифратор передает логическое значение 1, а на остальные — значение 0. Существует еще один очень полезный класс селекторных схем, предназначенный для выбора одного из n входов данных, значение которого передается на выход схемы. Выбор осуществляется на основе значений, поданных на так называемые входы выбора. Такие схемы называются *мультиплексорами*. Пример мультиплексорной схемы приведен на рис. А.38.



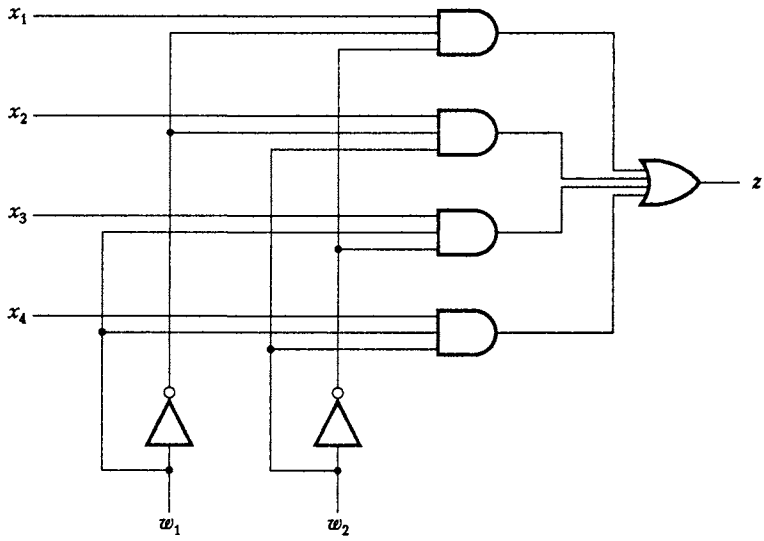


Рис. А.38. Четырехвходовый мультиплексор

У данной схемы два входных сигнала выбора — w_1 и w_2. Четыре возможные комбинации их значений используются для выбора одного из входов данных (x_1, x_2, x_3 или x_4), значение которого передается на выход z. Очевидно, такую же структуру будут иметь и большие мультиплексоры, в которых k входных сигналов выбора используются для соединения одного из 2^k входов данных с выходом. Типичной областью применения мультиплексоров является фильтрация данных, поступающих из множества разных источников. В частности, с помощью шестнадцати четырехвходовых мультиплексоров можно реализовать загрузку 16-разрядного регистра данных из одного из четырех источников.

Еще мультиплексоры используются в качестве базовых элементов для реализации логических функций. Для примера рассмотрим функцию f, определяемую таблицей истинности, приведенной на рис. А.39. Чтобы упростить эту функцию, переменные x_1 и x_2 следует рассматривать отдельно, как показано на рисунке. Обратите внимание, что для каждой пары значений переменных x_1 и x_2 значение функции соответствует одному из четырех термов: 0, 1, x_3 или $\bar{x}_3$. Это означает, что функцию можно реализовать с помощью четырехвходовой мультиплексорной схемы, где переменные x_1 и x_2 используются для выбора одного из четырех входных сигналов. Далее, если на входы данных подаются значения 0, 1, x_3 или $\bar{x}_3$, то, согласно таблице истинности, на выход мультиплексора передается значение, соответствующее функции f. Это универсальный подход. Любую функцию трех переменных можно реализовать с помощью одного четырехвходового мультиплексора. Любую функцию четырех переменных подобным же образом можно реализовать с помощью одного восьмивходового мультиплексора и т. д.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

⇒

x_1	x_2	f
0	0	0
0	1	x_3
1	0	1
1	1	$\bar{x}_3$

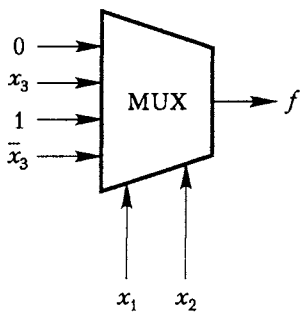


Рис. А.39. Реализация логической функции на основе мультиплексора

А.11. Программируемые логические устройства

В разделах А.2 и А.3 было показано, как можно любую логическую функцию представить в виде суммы произведений и реализовать с помощью схемы на основе вентилей И и ИЛИ. В разделе А.10 рассказывалось о реализации логической функции с применением мультиплексора. Теперь же речь пойдет об еще одном классе схем, обычно используемом для этой же цели. Описанные здесь схемы состоят из массивов логических элементов, которые для получения заданной сумы произведений можно программировать. Такие схемы называются ПЛУ — *программируемыми логическими устройствами* (Programmable Logic Device, PLD).

Блок-схема программируемого логического устройства показана на рис. А.40. У него n входных переменных (x_1, ..., x_n) и m выходных функций (f_1, ..., f_m). Каждая функция f_i реализуется как сумма произведений входных переменных. Значения переменных x_1, ..., x_n в исходной форме и в форме дополнений подаются на входы матрицы И, где из них формируется k термов-произведений. Оттуда они передаются в матрицу ИЛИ, где формируются выходные функции. В этом разделе описываются два наиболее распространенных типа программируемых логических устройств.

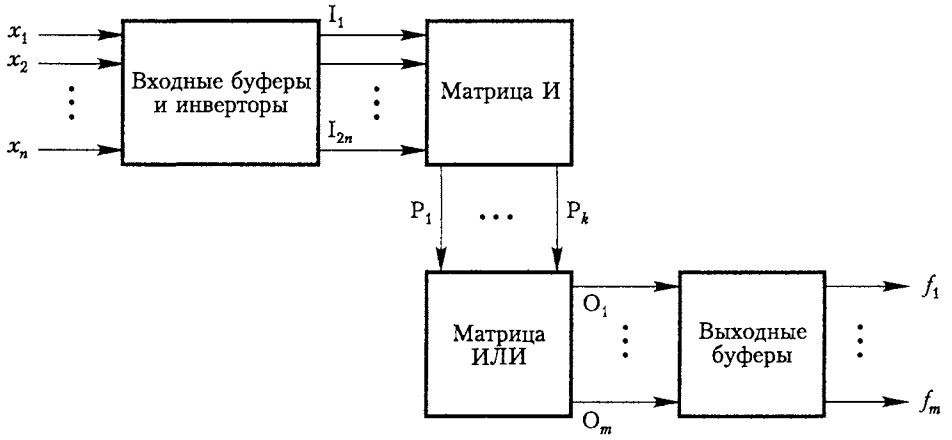


Рис. А.40. Блок-схема ПЛУ

А.11.1. Программируемая логическая матрица

Схема, в которой соединения с массивами И и ИЛИ можно программировать, называется *программируемой логической матрицей*, ПЛМ (Programmable Logic Array, PLA). Структура такой схемы представлена на рис. А.41. Программируемые соединения должны быть спроектированы таким образом, чтобы при отсутствии соединения с заданным входом вентиля И схема вела себя так, словно на этот вход подано значение 1 (следовательно, вход не будет влиять на произведение, вычисляемое данным вентилем). Аналогичным образом, при отсутствии соединения с заданным входом вентиля ИЛИ этот вход не должен воздействовать на выход вентиля (как если бы на него было подано логическое значение 0).

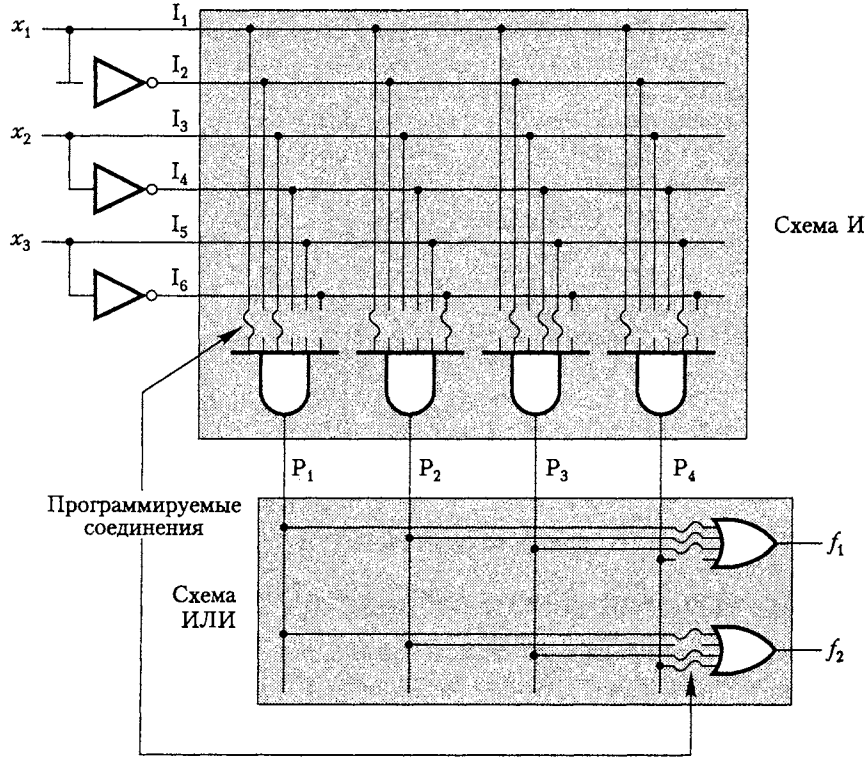
Программируемые соединения могут быть реализованы разными способами. Один из них состоит в том, чтобы расплавить перемычки в тех точках, где соединения не требуются. Для этого через определенные точки пропускается сильный ток. Кроме того, можно использовать транзисторные переключатели, управляемые стираемыми элементами памяти (см. раздел 5.3).

Простая программируемая логическая матрица, показанная на рис. А.41, может генерировать до четырех термов-произведений на основе трех входных переменных. С их помощью могут быть реализованы две выходные функции. Некоторые термы могут использоваться в нескольких функциях. В данном случае ПЛМ сконфигурирована для реализации двух функций:

$$f_1 = x_1x_2 + x_1\bar{x}_3 + \sim x_1\bar{x}_2x_3$$

$$f_2 = x_1x_2 + x_1x_3 + \bar{x}_1\bar{x}_2x_3$$

Для этих функций достаточно четырех термов, так как два из них будут использоваться обеими функциями. Реальные же ПЛМ имеют гораздо большие размеры.



$$f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$$

$$f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$$

Рис. А.41. Функциональная структура ПЛМ

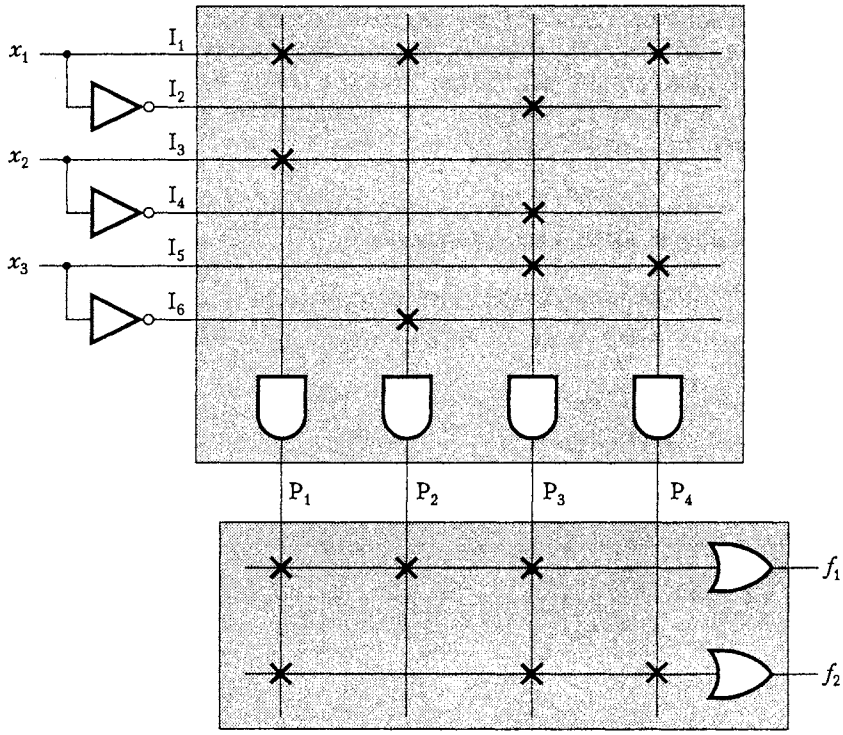


Рис. А.42. Упрощенная схема ПЛМ, показанной на рис. А.41

Хотя представленная выше схема прекрасно отражает базовые принципы функционирования ПЛМ, для описания больших матриц она неудобна. В технической литературе суммы и произведения нескольких переменных обычно обозначают значками с одним символическим входом. Линия, ведущая к этому входу, помечается крестиком ×, указывающим на программируемое подключение. Это соглашение принято и для рис. А.42, где приведена та же схема, что и на рис. А.41. В общем случае соединение может быть создано в любой точке пересечения вертикальной и горизонтальной линий, что позволяет реализовать разные функции заданных входных переменных.

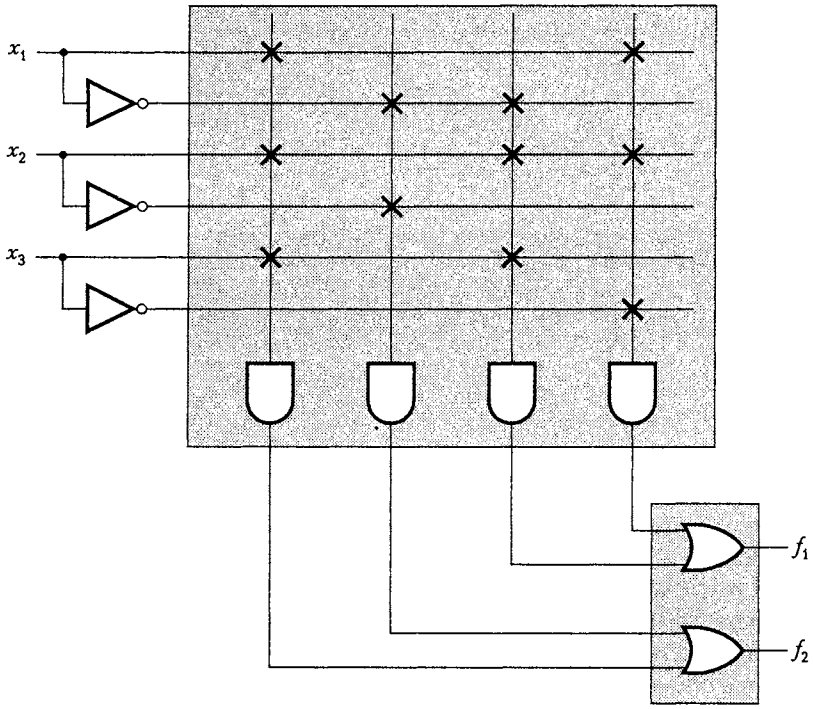
Структура ПЛМ очень эффективна с точки зрения занимаемой ею площади на чипе интегральной микросхемы. Поэтому подобные структуры часто используются для реализации управляющих схем на чипах процессоров. В таком случае желаемые подключения создаются на последней стадии производственного процесса, а не после его завершения.

А.11.2. Программируемая матричная логика

В программируемой логической матрице программируются обе составляющие: и матрица И, и матрица ИЛИ. На практике очень популярны похожие схемы, в которых входы матрицы И являются программируемыми, а соединения с матрицей

ИЛИ фиксированными. Такие устройства называются *микросхемами с ПМЛ* или *программируемой матричной логикой* (Programmable Array Logic, PAL).

На рис. А.43 показан простой пример схемы с ПМЛ, реализующей две функции. Количество вентилей И, соединенных с каждым вентилем ИЛИ, определяет максимальное число произведений, которое может входить в состав представления данной функции (в виде суммы произведений). Вентили И жестко соединены с конкретными вентилями ИЛИ. Это значит, что произведение не может использоваться в нескольких выходных функциях.



$$f_1 = x_1x_2\bar{x}_3 + \bar{x}_1x_2x_3$$

$$f_2 = \bar{x}_1\bar{x}_2 + x_1x_2x_3$$

Рис. А.43. Пример схемы с ПМЛ

Микросхемы ПМЛ выпускаются в разных конфигурациях. В них может быть много входных переменных и выходов, что позволяет на их основе реализовать очень сложные функции. Для большей гибкости на выходе вентилей ИЛИ в них включают триггеры. Таким образом, на одном чипе ПМЛ может быть реализована достаточно сложная логическая схема.

Насколько гибкими могут быть схемы на основе ПМЛ, можно судить по рис. А.44. Здесь вычисляется значение функции *f*, а затем мультиплексор определяет, какое ее значение — истинное, дополненное либо сохраненное (с предыду-

щего такта) — следует передать на выход. Входные сигналы выбора в мультиплексоре можно реализовать как программируемые соединения. Результирующее значение передается на выход через повторитель с тремя состояниями, управляемый сигналом, который разрешает выдачу значения. Обратите внимание, что выходной сигнал мультиплексора может использоваться как входной сигнал для термов-произведений или других вентилей ИЛИ в этой же ПМЛ-микросхеме. Таким образом могут создаваться схемы с несколькими уровнями логических вентилей.

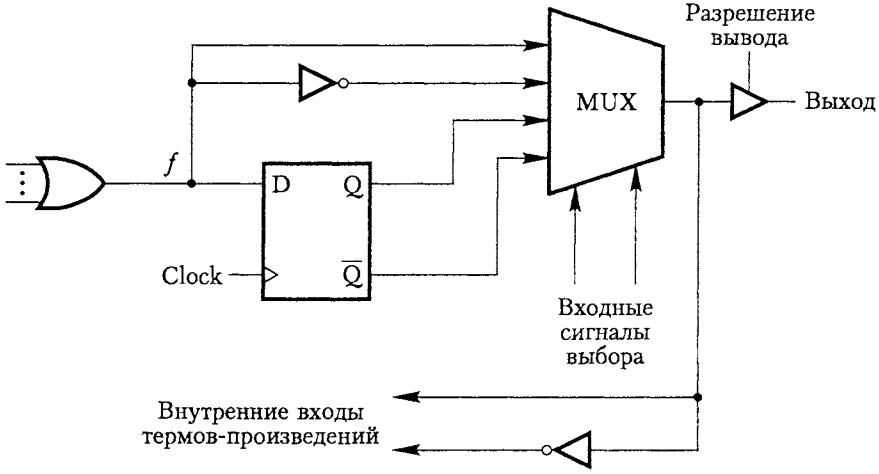


Рис. А.44. Пример обработки выходного сигнала ПМЛ-элемента

А.11.3. Сложные программируемые логические устройства

Микросхемы ПМЛ, безусловно, очень полезны, но они вмещают относительно небольшое количество логических элементов, поэтому для реализации типичной цифровой системы может потребоваться очень много таких чипов. Для подобных целей имеет смысл применять более крупные схемы, называемые *сложными программируемыми логическими устройствами, СПЛУ* (Complex Programmable Logic Device, CPLD). Они состоят из двух или нескольких блоков типа ПЛМ с программируемыми межсоединениями. Структуру такого чипа можно видеть на рис. А.45. Каждый блок типа ПМЛ соединен со множеством входных и выходных выводов. Соединения между блоками устанавливаются путем программирования переключателей, связанных с соединительными линиями. Коммутационный блок представляет собой набор горизонтальных и вертикальных проводящих линий. Каждую горизонтальную линию путем программирования соответствующих переключателей можно соединить с одной из вертикальных. Обычно схема не позволяет соединять любые вертикальные линии с любыми горизонтальными, поскольку для этого потребовалось бы слишком много переключателей. В реальных схемах их бывает намного меньше.

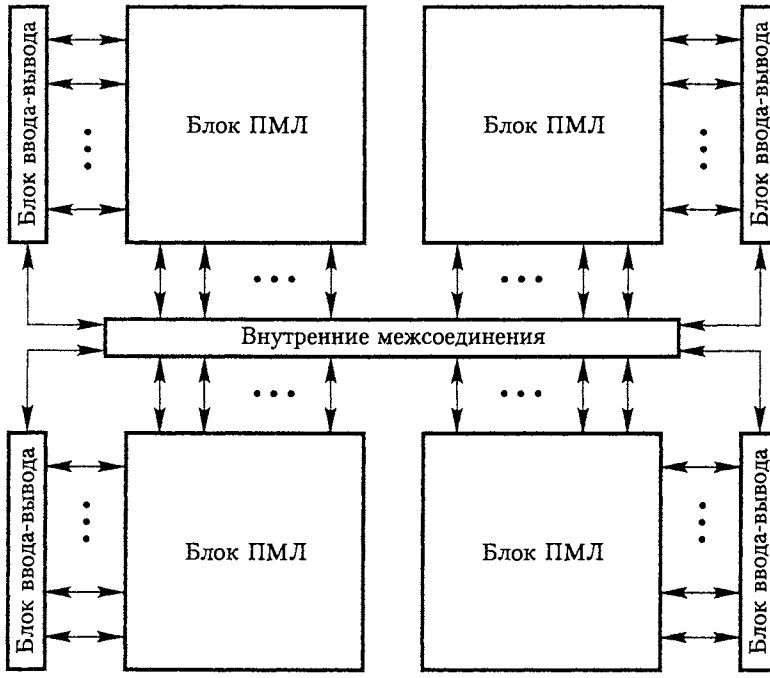


Рис. А.45. Структура сложного программируемого логического устройства

Серийно выпускаемые СПЛУ имеют разные размеры — они могут содержать от двух до более чем ста блоков ПМЛ. Такой чип программируется путем загрузки управляющей информации через *порт JTAG*. Этот четырехконтактный порт соответствует стандарту IEEE, разработанному Joint Test Automation Group.

А.12. Программируемые вентильные матрицы

Микросхемы ПМЛ достаточно универсальны, но их размер ограничен по той причине, что для каждой суммы произведений требуется один выходной контакт. Для преодоления этого ограничения разработан класс более мощных программируемых устройств, называемых *программируемыми вентильными матрицами, ПВМ* (Field Programmable Gate Array, FPGA). Концептуальная схема такой матрицы показана на рис. А.46. Она состоит из набора логических блоков (обозначенных черными квадратами), которые могут соединяться с помощью общих межсоединений. *Межсоединения*, выделенные на рисунке серым цветом, состоят из отрезков проводящих линий и программируемых переключателей. Переключатели используются для установки соединений между разными сегментами проводящих линий. Построенный таким образом чип обладает исключительной гибкостью. Для доступа к выходным и входным контактам чипа здесь предназначены буферы ввода-вывода.

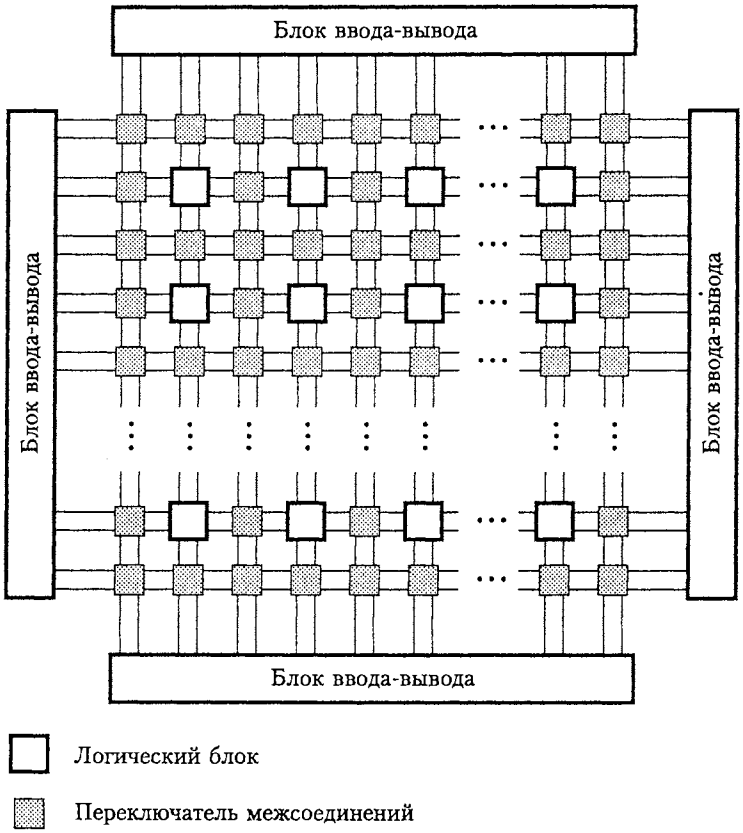


Рис. А.46. Концептуальная схема программируемой вентильной матрицы

Существует множество конструкций логических блоков и структур межсоединений. Логическими блоками могут служить простейшие схемы на основе мультиплексоров, реализующие логические функции (см. раздел А.10). В еще одной популярной конструкции в качестве логического блока используется простейшая таблица для выбора значения. Например, таблица с четырьмя входами может быть реализована в виде 16-разрядной схемы памяти, в которой хранится таблица истинности логической функции. Каждый бит памяти соответствует одной комбинации значений входных переменных. Путем программирования этой таблицы можно реализовать любую функцию четырех переменных. Еще логические блоки могут содержать триггеры, обеспечивающие дополнительную гибкость. В дополнение к логическим блокам многие программируемые вентильные матрицы включают достаточно большое количество ячеек памяти (на рис. А.46 они не показаны), которые могут использоваться для реализации таких структур, как очереди или RAM- и ROM-компоненты в приложениях для встроенных процессоров (см. главу 9).

С точки зрения пользователя, между программируемыми вентильными матрицами и сложными программируемыми логическими устройствами имеются существенные различия. Функциональные возможности чипов вентильных матриц

гораздо шире, и на их основе могут быть реализованы довольно большие и сложные логические схемы. Один такой чип может содержать более миллиона логических вентилей. Различна и скорость работы этих двух типов микросхем. Поскольку во внутренних межсоединениях вентильной матрицы используются программируемые переключатели, задержка на распространение сигнала в ней неизбежно будет более заметной, нежели у таких менее гибких устройств, как микросхемы ПМЛ или СПЛУ.

Программируемые вентильные матрицы становятся все более популярными у конструкторов цифровых логических устройств, поскольку они позволяют располагать на одном чипе очень сложные схемы и избегать разработки и создания пользовательских СБИС, что, во-первых, дорого, а во вторых, требует очень много времени. Пользуясь средствами автоматизированного проектирования, конструкторы могут создавать интегральные схемы на основе программируемых вентильных матриц не за месяцы, как пользовательские СБИС, а за считанные дни. Даже самые большие ИС на основе программируемых вентильных матриц стоят всего несколько сотен долларов, и затраты, связанные с их разработкой, очень малы по сравнению с затратами на разработку пользовательских чипов.

Основы программируемых логических устройств освещаются во многих современных книгах по проектированию логических схем. За более подробной информацией об этих устройствах читателя можно отослать и к документации от производителей.

А.13. Последовательные схемы

Логические схемы подразделяются на два класса: комбинаторные и последовательные. Выходной сигнал комбинаторной схемы полностью определяется текущими значениями на ее входах. Примерами таких схем могут служить дешифраторы и мультиплексоры, о которых рассказывалось в разделах А.9 и А.10. Выход *последовательной схемы* зависит не только от текущих значений на ее входах, но и от последовательности предыдущих входных значений. Такие схемы могут находиться в разных *состояниях*, что зависит от того, какой была последовательность предшествующих входных данных. В разделах А.7 и А.8 вам встречались два типа таких схем, а именно сдвиговые регистры и счетчики. Ниже мы приведем еще несколько примеров последовательных схем, рассмотрим их общую структуру и базовые принципы разработки.

А.13.1. Пример счетчика с прямым/обратным счетом

На рис. А.35 показана конфигурация счетчика с прямым счетом (суммирующего счетчика), реализованного посредством трех триггеров. Этот счетчик считает в таком порядке: 0, 1, 2, ..., 7, 0, ... Для обратного счета, то есть счета в порядке 0, 7, 6, ..., 1, 0, ..., применяются похожие схемы (см. упражнение А.26). При построении этих простых схем используется заложенная в Т-триггере возможность переключения состояний.

Сейчас же мы рассмотрим возможность реализации счетчиков на основе D-триггеров. В частности, речь пойдет о структуре счетчика, который способен считать как в прямом, так и в обратном порядке, в зависимости от значения на внешнем управляющем входе. Для того чтобы приводимый пример не оказался слишком большим, мы ограничимся счетом по модулю 4, для которого достаточно двух битов состояния, представляющих четыре возможных значения счетчика. Счетчик будет сконструирован на основе стандартной технологии синтеза последовательных схем. Такая схема считает в прямом направлении, если входной сигнал x равен 0, и в обратном, если входной сигнал равен 1. Текущее значение счетчика будет изменяться на отрицательном фронте тактового сигнала. Предположим, что нас особо интересует состояние счетчика в тот момент, когда мы досчитаем до 2. Поэтому при значении счетчика 2 мы будем выдавать выходной сигнал $z = 1$, а все остальное время сигнал z будет равен 0.

Такой счетчик можно реализовать как последовательную схему. Для того чтобы определить новое значение счетчика, которое будет установлено после очередного тактового импульса, достаточно знать значение x и *текущее значение* счетчика. Предшествующие его значения для этого не нужны. Если текущее значение равно 2 и $x = 0$, следующим значением будет 3. И не важно, получено текущее значение путем прямого счета от 1 или путем обратного счета от 3.

Прежде чем показать, как реализуется такая схема, давайте опишем ее поведение с помощью диаграммы состояний. У нашего счетчика имеется четыре разных состояния: S0, S1, S2 и S3. *Диаграмма состояний* — это граф, в котором состояния представлены окружностями (иногда называемыми узлами). Переходы между состояниями представлены стрелками с надписями. Связанная со стрелкой надпись указывает, какое значение переменной x вызывает данный переход и какое значение счетчика получается в результате. Диаграмма состояний для нашего счетчика с прямым/обратным счетом приведена на рис. А.47. Например, стрелка, исходящая от состояния S1 (счетчик = 1) при $x = 0$, указывает на состояние S2. При этом сообщается, что пока схема находится в состоянии S1 и $x = 0$, выходное значение z должно быть равным 0. Стрелка, ведущая от узла S2 к узлу S3, указывает, что когда $x = 0$, на следующем такте будет выполнен переход в состояние S3, а выходное значение z будет установлено в 1.

Обратите внимание, что диаграмма состояний описывает функциональное поведение счетчика вне какой бы то ни было связи с его конкретной реализацией. Так, рис. А.47 может соответствовать цифровой схеме, механическому счетному устройству или компьютерной программе. Подобные диаграммы позволяют описывать любую систему с последовательным поведением.

В качестве альтернативы диаграмме состояний для представления той же информации можно воспользоваться *таблицей состояний*. Такая таблица для нашего примера приведена на рис. А.48. В ней описаны переходы из каждого текущего состояния в *следующие состояния*, определяемые входным значением x. Выходной сигнал z определяется текущим состоянием схемы и входным значением x.

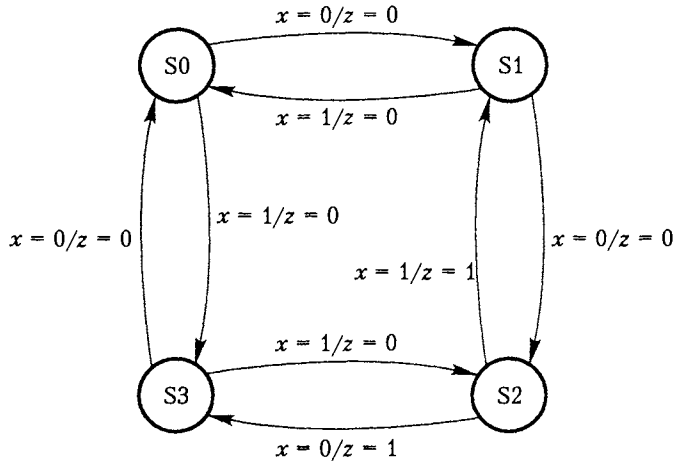


Рис. А.47. Диаграмма состояния для счетчика с прямым/обратным счетом по модулю 4, сигнализирующего о значении 2

Текущее состояние	Следующее состояние		Выход z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
S0	S1	S3	0	0
S1	S2	S0	0	0
S2	S3	S1	1	1
S3	S0	S2	0	0

Рис. А.48. Таблица состояний для счетчика с прямым/обратным счетом

Определив функциональное поведение счетчика, можно переходить к его физической реализации. Для того чтобы закодировать все четыре состояния счетчика, достаточно двух битов. Обозначим их как y_2 (старший бит) и y_1 (младший бит). Состояния считчика, определяемые значениями переменных y_2 и y_1, мы будем записывать в форме y_2y_1. У нашего счетчика четыре состояния y_2y_1: S0 = 00, S1= 01, S2 = 10 и S3= 11. Как видите, это просто двоичная запись чисел из диапазона от 0 до 3, применяемая для обеспечения наглядности счета. Переменные y_2 и y_1 называются *переменными состояния последовательной схемы*. При указанных значениях этих переменных таблицу состояний для нашего примера можно переписать так, как показано на рис. А.49. Для обозначения следующих состояний в ней используются переменные Y_2 и Y_1.

Мы могли бы выбрать и другие обозначения состояний y_2y_1. В частности, они могут быть такими: S0 = 10, S1= 11, S2 = 01 и S3= 00. Но такие обозначения не совсем логичны, хотя схема прекрасно бы функционировала. Реализация различных состояний обычно требует разных затрат (см. упражнение А.32).

Мы планировали создать нашу схему на основе D-триггеров, в которых значения двух переменных состояния будут сохраняться между последовательными

тактовыми импульсами. Выход Q каждого триггера будет представлять переменную текущего состояния y_i, а вход D — переменную следующего состояния Y_i. Заметьте, что Y_i является функцией переменных y_2, y_1 и x (это хорошо видно на рис. А.49).

Текущее состояние	Следующее состояние		Выход z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
$y_2\,y_1$	$Y_2\;Y_1$	$Y_2\;Y_1$		
0 0	0 1	1 1	0	0
0 1	1 0	0 0	0	0
1 0	1 1	0 1	1	1
1 1	0 0	1 0	0	0

Рис. А.49. Значения состояний для примера, приведенного на рис. А.48

На основе представленной здесь таблицы для функций Y_2 и Y_1 можно составить вот такие выражения:

$$Y_2 = \bar{y}_2 y_1 \bar{x} + y_2 \bar{y}_1 \bar{x} + \bar{y}_2 \bar{y}_1 x + y_2 y_1 x = y_2 \oplus y_1 \oplus x$$

$$Y_1 = \bar{y}_2 \bar{y}_1 \bar{x} + y_2 \bar{y}_1 \bar{x} + \bar{y}_2 \bar{y}_1 x + y_2 \bar{y}_1 x = \bar{y}_1$$

Выходное значение z определяется следующим образом:

$$z = y_2 \bar{y}_1$$

С учетом этих выражений составляется схема, приведенная на рис. А.50.

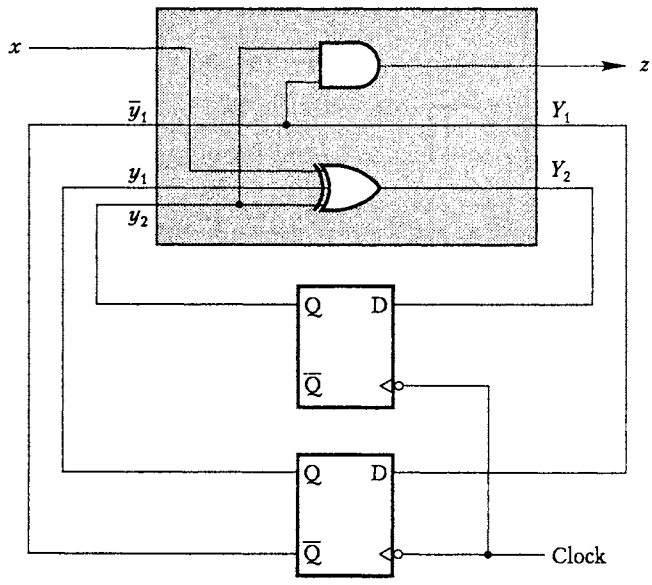


Рис. А.50. Реализация счетчика с прямым/обратным счетом

А.13.2. Временные диаграммы

Для того чтобы полнее представить себе работу схемы счетчика, необходимо рассмотреть его временную диаграмму. На рис. А.51 приведен пример возможной последовательности событий, происходящих в нашей схеме. Предполагается, что переходы между состояниями (изменения значений триггеров) происходят на отрицательном фронте тактирующего сигнала и что начальным состоянием счетчика является S0. Вначале $x = 0$. Поэтому в момент времени t_0 счетчик переходит в состояние S1, затем, в момент времени t_1, он переходит в состояние S2 (и на выходе z появляется значение 1), а затем, в момент времени t_2, — в состояние S3 (на выходе z мы снова видим 0). Далее, в момент времени t_3, счетчик снова переходит в состояние S0. Предположим, что в этот момент входное значение x меняется на 1, из-за чего счетчик начинает считать в обратном порядке. Когда в момент времени t_5 счетчик снова достигнет состояния S2, на выходе z опять появится значение 1

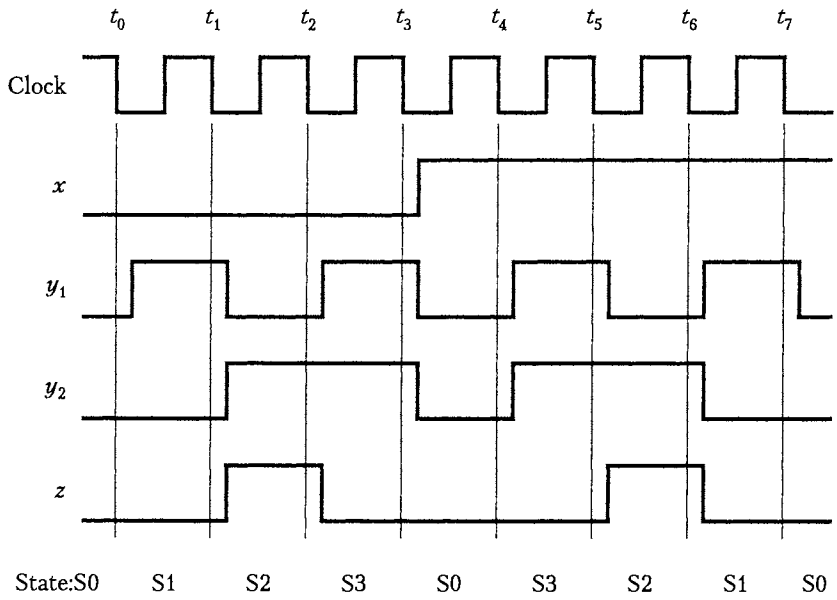


Рис. А.51. Временная диаграмма для схемы, приведенной на рис. А.50

Обратите внимание, что все изменения сигналов происходят сразу после появления отрицательного фронта тактового сигнала, и сигналы больше не меняются до тех пор, пока не появится отрицательный фронт следующего тактового сигнала. Время задержки между фронтом тактового сигнала и изменением переменной y_i — это время задержки на распространение сигнала в триггерах, на основе которых создана схема счетчика. Важно отметить, что вход x управляется тем же тактовым сигналом и изменяется в начале такта. Схемы, в которых все изменения управляются тактовым сигналом, называются *синхронными последовательными схемами.*

Еще одно важное наблюдение касается соответствия диаграммы состояний, представленной на рис. А.47, временной диаграмме. Возьмем, к примеру, тактовый

период между моментами времени t_1 и t_2. В течение этого времени схема находится в состоянии S2 и на ее входе сохраняется значение $x = 0$. На диаграмме состояний эта ситуация представлена исходящей из узла S2 стрелкой с надписью $x = 0$. Поскольку эта стрелка указывает на узел S3, на следующем фронте тактового сигнала (t_2) переменные y_1 и y_2 принимают значения, соответствующие состоянию S3. А пока счетчик будет оставаться в состоянии S2, на выходе z будет значение 1.

А.13.3. Модель конечного автомата

Рассмотренный нами пример счетчика с прямым/обратным счетом, реализованного в виде синхронной последовательной схемы на основе триггеров и вентильной комбинаторной логики (рис. А.50), легко обобщить до формальной модели *конечного автомата*, показанной на рис. А.52. В этой модели запаздывание при прохождении сигнала через элементы задержки равно длительности такта. Это время между изменениями, происходящими в точках Y_i, и соответствующими изменениями в точках y_i. В данной модели предполагается, что через блок комбинаторной логики сигналы проходят без задержки; поэтому выходные значения z, Y_1 и Y_2 являются мгновенными функциями входных значений x, y_1 и y_2. В реальной схеме все элементы, как показано на рис. А.51, служат источниками определенных задержек. Схема будет работать правильно лишь при условии, что задержка блока комбинаторной логики не велика по сравнению с длительностью такта. Значения на выходах следующего состояния, Y_1 и Y_2, должны быть установлены таким образом, чтобы состояния триггеров изменились до окончания такта. Кроме того, в течение некоторого времени на выходе z требуемого значения может не быть, но оно должно появиться задолго до окончания такта.

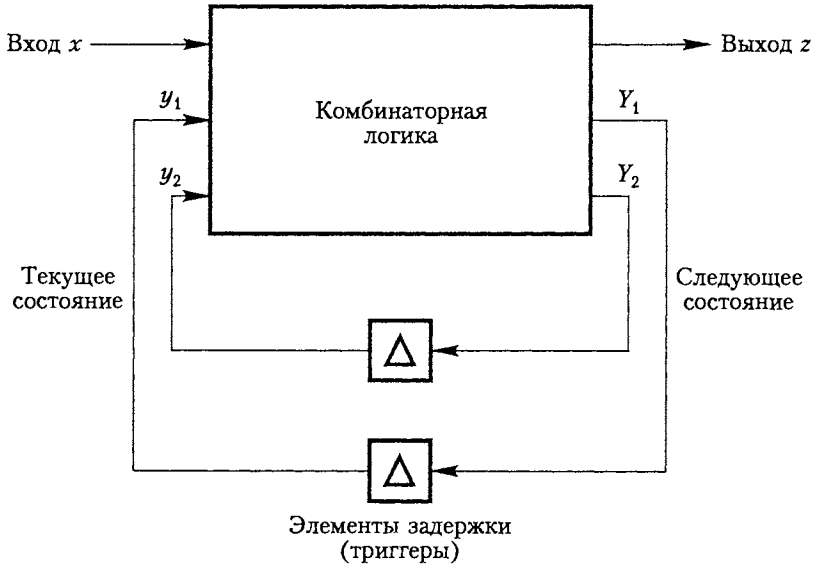


Рис. А.52. Модель конечного автомата

Входы блока комбинаторной логики соединены с выходами триггеров y_i, представляющих текущее состояние, и с внешним входом x. Выходы блока, обозначенные как Y_i, соединены со входами триггеров и внешним выходом z. На отрицательном фронте тактового сигнала, соответствующем концу текущего такта, в триггеры загружаются значения выходов Y_i, которые становятся следующими значениями переменных состояния y_i. Поскольку выходы триггеров соединены со входами блока комбинаторной логики, они вместе со следующим значением внешнего входа x определяют новые значения переменных z и Y_i. На следующем такте новые значения Y_i передаются на входы y_i, и процесс повторяется сначала. Таким образом, триггеры образуют обратную связь между выходом и входом комбинаторного блока с задержкой в один такт.

Хотя в рассмотренной нами схеме (рис. А.52) имелся лишь один внешний вход и две переменные состояния, совершенно очевидно, что в схеме может быть и по нескольку переменных каждого из трех типов.

А.13.4. Синтез конечных автоматов

Теперь давайте подытожим сказанное и определим общую последовательность создания представленной на рис. А.52 синхронной схемы, в основу которой положена диаграмма состояний, показанная на рис. А.47. Процесс можно разбить на следующие ключевые этапы.

1. Составление диаграммы или таблицы состояний.
2. Определение количества и выбор подходящего типа триггеров.
3. Определение значений, которые будут храниться в триггерах по каждому из состояний, указанных в диаграмме состояний (этот этап называется назначением состояний).
4. Составление таблицы состояний.
5. Составление таблицы истинности для блока комбинаторной логики.
6. Разработка схемы реализации блока комбинаторной логики.

Пример

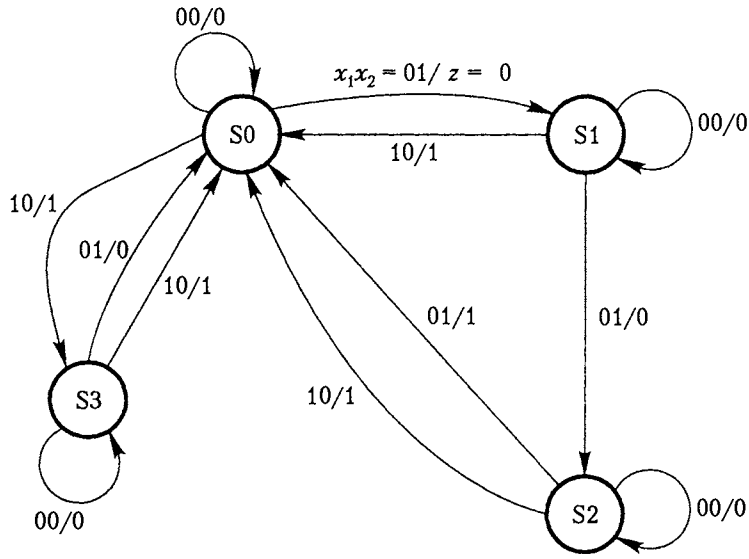
В качестве еще одного примера конечного автомата со входами и выходами будет рассмотрен торговый автомат, принимающий монеты и выдающий некоторый товар. Для того чтобы упростить изложение, предположим, что этот автомат принимает только монеты достоинством 25 и 10 центов. Автомат принимает монеты до тех пор, пока не получится сумма в 30 или более центов, после чего выдает товар. Причем, получив больше 30 центов, автомат сдачи не дает. Пусть опускаемые в автомат монеты представляют две двоичные переменные, x_1 и x_2. Монете достоинством в 25 центов соответствует значение $x_1 = 1$ ($x_2 = 0$), а монете достоинством в 10 центов — значение $x_2 = 1$ ($x_1 = 0$). Монеты опускаются по одной, так что комбинации $x_1x_2 = 11$ быть не может. Пусть двоичная выходная переменная z обозначает выдачу автоматом товара, то есть $z = 0$, когда товар не выдается, и $z = 1$ в противном случае.

При синтезе логической схемы такого автомата первым делом нужно построить диаграмму или составить таблицу состояний. Имеет смысл для каждого из состояний составить словесное описание, чтобы позднее легче было определить, сколько триггеров потребуется для представления нужного количества состояний. Состояния определяют общую сумму денег, опущенных в автомат на данный момент. Поскольку монеты могут опускаться в любом порядке до тех пор, пока их общая сумма не составит или не превысит 30 центов, нам потребуются следующие четыре состояния:

- S0 — ничего не опущено (начальное состояние);
- S1 — 10 центов;
- S2 — 20 центов;
- S3 — 25 центов.

Другие состояния не потребуются, поскольку по достижении состояния S2 или S3 любая следующая монета дополнит сумму до необходимого максимума, после чего будет сгенерировано выходное значение $z = 1$ и автомат вернется в состояние S0, то есть будет готов к следующей операции продажи.

Диаграмма состояний, описывающая поведение нашего торгового автомата, приведена на рис. А.53.



$x_1 = 1$ — опущена монета в 25 центов
$x_2 = 1$ — опущена монета в 10 центов
$z = 1$ — выдача товара (получена сумма в 30 или более центов)

Рис. А.53. Диаграмма состояний торгового автомата

Обратите внимание, что в диаграмме отсутствует входная пара значений $x_1x_2 = 11$, поскольку опустить в автомат обе монеты одновременно невозможно. Кроме того, как видите, из каждого узла-состояния исходит стрелка, указывающая на него же и обозначенная как 00/0. Эта стрелка говорит о том, что пока в автомат не опущена ни одна монета, он остается в прежнем состоянии.

Описанный автомат может иметь всего четыре состояния, для реализации которых достаточно двух триггеров. Обозначим их как y_2 и y_1 и назначим им следующие значения: S0 = 00, S1 = 01, S2 = 10 и S3 = 11. Результирующая таблица состояний приведена на рис. А.54. Прочерки в таблице соответствуют комбинации $x_1x_2 = 11$, которая на практике невозможна. Эти элементы, являющиеся безразличными значениями, при разработке схемы будут нам очень полезны, о чем вы узнаете позднее.

	Текущее состояние	Следующее состояние				Выход z			
		$x_1x_2 = 00$	$x_1x_2 = 01$	$x_1x_2 = 10$	$x_1x_2 = 11$	$x_1x_2 = 00$	$x_1x_2 = 01$	$x_1x_2 = 10$	$x_1x_2 = 11$
	$y_2\ y_1$	$Y_2\ Y_1$	$Y_2\ Y_1$	$Y_2\ Y_1$	$Y_2\ Y_1$				
S0	0 0	0 0	0 1	1 1	-	0	0	0	-
S1	0 1	0 1	1 0	0 0	-	0	0	1	-
S2	1 0	1 0	0 0	0 0	-	0	1	1	-
S3	1 1	1 1	0 0	0 0	-	0	1	1	-

Рис. А.54. Таблица состояний для торгового автомата

Этим завершается первый этап процесса разработки автомата. Составленной нами таблице состояний соответствует таблица истинности, приведенная на рис. А.55, — она определяет функцию, которую нам предстоит реализовать в блоке комбинаторной логики. Из таблицы легко вывести следующие выражения:

$$Y_2 = \bar{x}_1\bar{x}_2y_2 + x_2\bar{y}_2y_1 + x_1\bar{y}_2\bar{y}_1$$
$$Y_1 = \bar{x}_1\bar{x}_2y_1 + \bar{y}_2\bar{y}_1(x_1 + x_2)$$
$$z = y_2(x_1 + x_2) + x_1y_1$$

Как видите, термы $\bar{x}_1\bar{x}_2$ и $(x_1 + x_2)$ встречаются в приведенных выражениях более одного раза. Это удешевляет реализацию логического блока.

Последовательные схемы удобнее всего реализовывать в виде ПМЛ, СПЛУ и программируемых вентильных матриц, поскольку все эти типы схем состоят из триггеров и логических вентилей. Современные средства автоматизированного проектирования позволяют синтезировать последовательные схемы непосредственно на основе спецификаций, составленных в терминах диаграммы состояний.

Обратите внимание на тот факт, что в таблице на рис. А.54 для состояний S2, S3 значения следующих состояний и выходной переменной одинаковы при всех входных комбинациях, в которых происходит изменение состояния. Это означает,

что для представления итоговых сумм в 20 и 25 центов двух разных состояний не требуется. Для них достаточно одного общего состояния, поскольку, как только в автомате окажется одна из этих двух сумм, поступление любой следующей монеты приведет к выдаче товара ($z = 1$) и возврату в состояние S0. Таким образом, состояния S2 и S3 эквивалентны и могут быть заменены одним состоянием. Это означает, что для реализации нашего торгового автомата достаточно трех состояний. Однако для них по-прежнему требуется два триггера. В общем случае, с уменьшением количества состояний обычно уменьшается и количество триггеров, а вся схема упрощается.

Источником еще одного упрощения может служить комбинаторная логика. От того, как вы обозначите состояния, зависит логическая спецификация, а разным спецификациям может соответствовать разное количество вентилей. Далее мы эту тему развивать не будем, но читатель может себе представить, как много интересных аспектов имеется у задачи оптимизации последовательных схем.

x_1	x_2	y_2	y_1	Y_2	Y_1	z
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	0	1	0	0	1
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	d	d	d
1	1	0	1	d	d	d
1	1	1	0	d	d	d
1	1	1	1	d	d	d

Рис. А.55. Спецификация комбинаторной логики для торгового автомата

Напоследок заметим, что для представления переменных состояния могут применяться и триггеры других типов. Мы использовали триггеры типа D, чтобы предельно упростить задачу. Более гибкие триггеры, и в частности JK, позволяют сократить объем необходимых логических схем. Соответствующие пояснения вы найдете в упражнениях А.35 и А.36. Рассказывая о последовательных схемах, мы опирались на схемы, управляемые тактовым сигналом. Но последовательную схему можно создать и без тактового входа. Такие схемы, называемые *асинхронными последовательными схемами*, разрабатывать несколько сложнее, чем синхронные последовательные.

А.14. Резюме

Основная задача настоящего приложения состояла в том, чтобы познакомить читателя с базовыми концепциями логического проектирования и в общих чертах описать основные конфигурации логических схем, используемых в компьютерных системах. Изучив данный материал, читатель сможет лучше понять архитектурные концепции, обсуждаемые в основной части книги. Как уже было сказано, проектирование логических схем выполняется с помощью средств автоматизированного проектирования (САПР), освобождающих конструктора от проработки множества деталей. В умелых руках такие средства могут стать исключительно эффективным инструментом.

Использование современных технологий изготовления интегральных схем и САПР способствовало революционизированию логического проектирования. В настоящее время выпускается широчайший ассортимент компонентов интегральных схем, цены на которые постоянно снижаются, тем не менее появляются все новые и новые разработки и усовершенствуются уже существующие. В этом приложении мы рассказали о некоторых базовых компонентах, используемых при разработке цифровых систем.

С точки зрения конструктора, важнейшими характеристиками интегральной схемы являются ее стоимость и быстродействие. Эти показатели улучшаются за счет предельного сокращения количества модулей ИС. Вместо множества отдельных модулей по возможности используются большие чипы, содержащие сложные логические схемы. В частности, во многих случаях эффективным решением является применение микросхем СПЛУ и программируемых вентильных матриц.

Существует еще два важных аспекта разработки логических схем, роль которых все более возрастает. Простота тестирования результирующей схемы важна, с одной стороны, для подтверждения корректности ее работы, а с другой — для ее восстановления в случае поломки. Кроме того, когда перед конструкторами стоит задача повышения надежности системы, разрабатываются дополнительные, в некоторых случаях даже избыточные логические схемы (например, дублирующие отдельные элементы системы). Как правило, решение подобных задач ведет к увеличению стоимости компонентов, и конструкторам приходится искать приемлемый компромисс между требуемыми характеристиками схем и их стоимостью.

Упражнения

А.1. Выразите функцию COINCIDENCE в форме суммы произведений, если COINCIDENCE = ИЛИ-НЕ.

А.2. Докажите следующие тождества с помощью алгебраических преобразований, а также с помощью таблиц истинности.

а) $\overline{a \oplus b} \oplus c = \bar{a}\bar{b}\bar{c} + ab\bar{c} + \bar{a}bc + a\bar{b}c$;

б) $x + w\bar{x} = x + w$;

в) $x_1\bar{x}_2 + \bar{x}_2x_3 + x_3\bar{x}_1 = x_1\bar{x}_2 + x_3\bar{x}_1$.

А.3. Выведите минимальные формы для суммы произведений функций f_1, f_2, f_3 и f_4 переменных x_1, x_2 и x_3, приведенных на рис. УА.1. Существует ли более одной минимальной формы каждой из этих функций? Если да, выведите все эти формы.

x_1	x_2	x_3	f_1	f_2	f_3	f_4
0	0	0	1	1	d	0
0	0	1	1	1	1	1
0	1	0	0	1	0	1
0	1	1	0	1	1	d
1	0	0	1	0	d	d
1	0	1	0	0	0	d
1	1	0	1	0	1	1
1	1	1	1	1	1	0

Рис. УА.1. Логические функции для упражнения А.3

А.4. Найдите минимальную форму суммы произведений для функции f, если:

$$f = x_1(x_2\bar{x}_3 + x_2x_3 + \bar{x}_2\bar{x}_3x_4) + x_2 x_4(\bar{x}_3 + x_1)$$

и

$$d = x_1\bar{x}_2(x_3x_4 + \bar{x}_3\bar{x}_4) + \bar{x}_1\bar{x}_3x_4$$

А.5. Имеется функция

$$f(x_1, \ldots, x_4) = (x_1 \oplus x_3) + (x_1x_3 + \bar{x}_1\bar{x}_3)\, x_4 + x_1\bar{x}_2$$

а) С помощью карты Карно найдите для этой функции сумму произведений с минимальной стоимостью.

б) Найдите сумму произведений с минимальной стоимостью для $\bar{f}$, то есть для дополнения функции f. Затем с помощью закона де Моргана образуйте дополнение этого выражения, чтобы получить выражение для f в форме суммы произведений. Сравните его стоимость со стоимостью выражения для вычисления суммы произведений, полученной при выполнении задания А.5, *а*. Можете ли вы сделать на основании данного результата какое-либо общее заключение?

А.6. Найдите выражение с минимальной стоимостью для функции $f(x_1, x_2, x_3, x_4)$, где $f = 1$, если одна или две входные переменные имеют логическое значение 1. В противном случае $f = 0$.

А.7. На рис. А.6 определены правила 4-разрядного двоичного кодирования десятичной цифры. Нарисуйте схему с четырьмя входами, помеченными как $b_3, \ldots, b_0$, и выходом f при условии, что $f = 1$, если 4-разрядный код соответствует одной из существующих десятичных цифр; в противном случае $f = 0$. Приведите реализацию этой схемы, которая имеет минимальную стоимость.

А.8. Два 2-разрядных числа, $A = a_1a_0$ и $B = b_1b_0$, должны сравниваться с помощью четырехразрядной функции $f(a_1, a_0, b_1, b_0)$. Функция f имеет значение 1, если

$$v(A) \leq v(B)$$

где $v(X) = x_1 \times 2^1 + x_0 \times 2^0$ для любого 2-разрядного двоичного числа. Предположим, что переменные A и B таковы, что $|v(A) - v(B)| \leq 2$. Синтезируйте f, используя минимум вентилей.

А.9. Повторите упражнение А.8 при условии, что $f = 1$, если

$$v(A) > v(B)$$

и действует входное ограничение

$$v(A) + v(B) \leq 4$$

А.10. Докажите, что ассоциативный закон не применим к оператору И-НЕ.

А.11. Реализуйте следующую функцию с помощью не более чем шести вентилей И-НЕ, каждый из которых имеет по три входа.

$$f = x_1x_2 + x_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3x_4) + \bar{x}_1\bar{x}_2x_3\bar{x}_4$$

Разрешается использовать дополнения входных переменных.

А.12. Покажите, как реализовать следующую функцию с помощью шести или менее 2-входовых вентилей И-НЕ. Дополнения входных переменных применять не разрешается.

$$f = x_1x_2 + \bar{x}_3 + \bar{x}_1x_4$$

А.13. Используя только вентили И-НЕ, реализуйте следующую функцию с минимальной стоимостью. Дополнения входных переменных применять не разрешается.

$$f = (x_1 + x_3)(\bar{x}_2 + \bar{x}_4)$$

А.14. Система кодирования чисел, согласно которой последовательные числа, представленные двоичными кодами, отличаются только одним битом, называется кодом Грея. Таблица истинности 3-разрядного кода Грея для конвертора двоичного кода приведена на рис. УА.2, *а*.

а) Реализуйте функции f_1, f_2 и f_3 с использованием только вентилей И-НЕ.

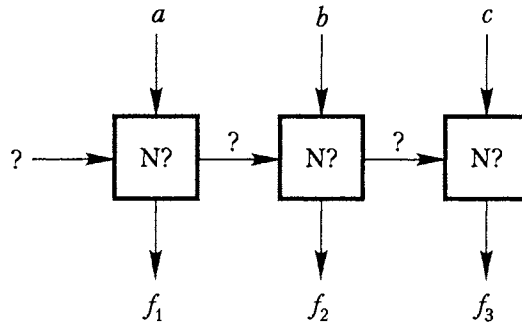
б) Чтобы минимизировать стоимость результирующей схемы, нужно выделить такие соответствия между входными и выходными переменными:

$$f_1 = a$$
$$f_2 = f_1 \oplus b$$
$$f_3 = f_2 \oplus c$$

Используя эти отношения, выделите ту часть схемы N, которую можно повторить, как показано на рис. УА.2, *б*. Сравните общее количество вентилей И-НЕ, необходимое для реализации операции преобразования в данной форме схемы, с количеством вентилей в схеме, которая у вас получилась в упражнении А14, *а*.

3-разрядный код Грея			Двоичный код		
a	b	c	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1

а



б

Рис. УА.2. Преобразование двоичных чисел в код Грея для упражнения А.14: 3-разрядный код Грея для преобразования двоичного кода (*а*); схема преобразования кода (*б*)

А.15. Реализуйте функцию Исключающее ИЛИ, используя только четыре 2-входовых вентиля И-НЕ.

А.16 На рис. А.37 показана схема декодера для 7-сегментного индикатора. Реализуйте данную таблицу истинности с использованием вентилей И, ИЛИ и НЕ. Проверьте, совпадают ли полученные результаты с представленными на указанном рисунке.

А.17. В логической схеме, представленной на рис. УА.3, вентиль 3 работает некорректно и генерирует на выходе F1 значение 1 независимо от его входных значений. Перестройте эту схему, предельно ее упростив, чтобы получилась эквивалентная схема с минимальным количеством вентилей. Повторно выполните задание, предполагая, что ошибка содержится в точке F2, где всегда генерируется логическое значение 0.

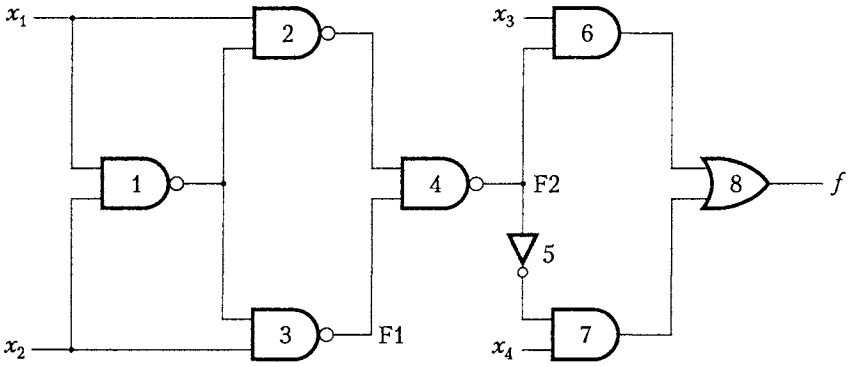


Рис. УА.3. Логическая схема, содержащая ошибку

А.18. На рис. А.16 показана структура универсальной микросхемы КМОП. Разработайте на ее основе КМОП-схему, реализующую такую функцию:

$$f(x_1, ..., x_4) = \bar{x}_1\bar{x}_2 + \bar{x}_3\bar{x}_4$$

Используйте как можно меньше транзисторов. (Подсказка: подумайте о последовательно-параллельных цепях транзисторов. В качестве образца рассмотрите последовательную, а также параллельную схемы понижающей и повышающей цепей (рис. А.17 и А.18).

А.19. Нарисуйте временную диаграмму для выхода Q в схеме JK-триггера, показанной на рис. А.31, используя временные диаграммы входных сигналов, которые приведены на рис. УА.4, и предполагая, что исходным состоянием триггера является 0.

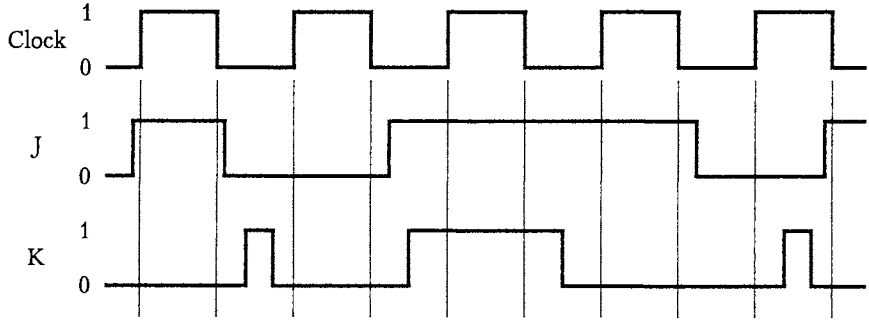


Рис. УА.4. Временные диаграммы входных сигналов в схеме JK-триггера

A.20. Составьте таблицу истинности для схемы на основе вентилей И-НЕ, которую вы видите на рис. УА.5. Сравните ее с таблицей истинности, приведенной на рис. А.24, *б*, а затем выясните, можно ли считать эквивалентными схемы, представленные на рис. А.25, *а* и А.26.

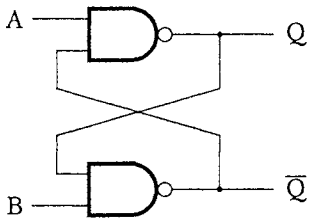


Рис. УА.5. Схема защелки И-НЕ

A.21. Определите время установки и хранения в сравнении со временем задержки вентиля ИЛИ-НЕ для D-триггера, тактируемого отрицательным фронтом сигнала (рис. А.29).

A.22. В схеме на рис. А.27, *а* замените все вентили И-НЕ вентилями ИЛИ-НЕ. Составьте таблицу истинности для результирующей схемы. Сравните эту схему со схемой на рис. А.27, *а*.

A.23. На рис. А.33 показана схема сдвигового регистра, перемещающего данные вправо на одну позицию за раз под воздействием тактового сигнала. Модифицируйте этот регистр, чтобы он мог сдвигать данные на одну или две позиции за раз под воздействием тактового сигнала и дополнительного управляющего входа ONE/TWO.

A.24. У 4-разрядного сдвигового регистра имеется два управляющих входных сигнала: INITIALIZE и RIGHT/LEFT. Когда на вход INITIALIZE подается значение 1, в регистр независимо от тактового сигнала загружается число 1000. Когда значение на входе INITIALIZE равно 0, сигналы на тактовом входе должны сдвигать содержимое регистра. Данные смещаются вправо или влево, когда значение RIGHT/LEFT равно соответственно 1 или 0. Нарисуйте схему такого регистра на основе D-триггеров со входами установки и очистки, как показано на рис. А.32.

A.25. Создайте схему 3-входового декодера с 8 выходами при условии, что в нем должны использоваться вентили, имеющие не более двух входов.

A.26. На рис. А.35 показан 3-разрядный счетчик прямого счета. Счетчик, считающий в порядке 7, 6, ..., 1, 0, 7, ..., называется счетчиком обратного счета. Счетчик, который может считать в любом из направлений в зависимости от сигнала UP/DOWN, называется счетчиком прямого/обратного счета. Приведите логическую схему трехразрядного счетчика прямого/обратного счета, который можно установить в любое из состояний путем параллельной загрузки его триггеров из внешнего источника. Управляющий вход LOAD/COUNT должен определять, какая операция — загрузка счетчика или счет — будет выполняться.

А.27. На рис. А.35 показан асинхронный 3-разрядный счетчик прямого счета. Спроектируйте 4-разрядный синхронный счетчик прямого счета, который считает в порядке 0, 1, 2, ..., 15, 0, ... Используйте для его создания Т-триггеры. В синхронном счетчике состояние всех триггеров меняется одновременно. Поэтому главный тактовый вход должен непосредственно соединяться с тактовыми входами всех триггеров.

А.28. Необходимо реализовать функцию-переключатель, определяемую следующим выражением:

$$f(x_1, x_2, x_3, x_4) = x_1x_3\bar{x}_4 + \bar{x}_1\bar{x}_3x_4 + \bar{x}_2\bar{x}_3\bar{x}_4$$

а) Как реализовать эту функцию в виде 8-входовой мультиплексорной схемы?

б) А можно ли реализовать данную функцию в виде 4-входовой мультиплексорной схемы? Если да, покажите как.

А.29. Выполните упражнение А.28 для функции

$$f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2x_3 + x_2x_3x_4 + \bar{x}_1\bar{x}_4$$

А.30. а) Каково общее количество разных функций $f(x_1, x_2, x_3)$ трех двоичных переменных?

б) Сколько этих функций можно реализовать в виде ПМЛ-схемы того же типа, что и представленная на рис. А.43?

в) Какое минимальное изменение в схеме на рис. А.43 позволит реализовать функцию трех переменных в виде единственной схемы ПМЛ?

А.31. Рассмотрим ПМЛ-схему, приведенную на рис. А.43. Предположим, что в нее добавлена четвертая входная переменная, x_4, недополненная и дополненная формы которой могут соединяться со всеми четырьмя вентилями И тем же способом, что и переменные x_1, x_2 и x_3.

а) Может ли эта модифицированная ПМЛ-схема использоваться для реализации такой функции:

$$f = x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$$

Если да, покажите как.

б) Сколько функций трех переменных нельзя реализовать с помощью этой ПМЛ-схемы?

А.32. Завершите разработку счетчика прямого/обратного счета, представленного на рис. А.47, используя значения состояний S0 = 10, S1 = 11, S2 = 01, S3 = 00. Сравните свою схему с приведенной в разделе А.13.1.

А.33. На основе D-триггеров спроектируйте 2-разрядный синхронный счетчик той же структуры, что и приведенный на рис. А.50, который бы считал в порядке ..., 0, 3, 1, 2, 0, ... У этой схемы нет внешних входов, а на ее выходы передаются значения триггеров.

A.34. Повторите задачу A.33 для 3-разрядного счетчика, считающего в порядке ..., 0, 1, 2, 3, 4, 5, 0, ... Для упрощения схемы используйте безразличные значения 6 и 7.

A.35. В разделе A.13 для создания синхронной последовательной схемы использовались D-триггеры. Это простейшее решение, поскольку значения логической функции для входа D определяются следующим состоянием, указанным в таблице состояний. Предположим, что вместо D-триггеров для создания той же схемы будут использованы JK-триггеры. Составив таблицу состояний, покажите, как определить значения для входов J и K в виде функции от каждого возможного перехода из текущего состояния триггера в следующее. (Подсказка: таблица должна содержать четыре строки, по одной для каждого перехода: $0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$, $1 \rightarrow 1$; значениями входов J и K должны быть 0, 1 или «безразлично»). Примените информацию этой таблицы для синтеза отдельных логических функций для обоих входов каждого из двух триггеров 2-разрядного двоичного счетчика из упражнения A.33. Как вы считаете, данная реализация проще или, наоборот, сложнее по сравнению с реализацией счетчика на основе D-триггеров?

A.36. Повторите упражнение A.34, используя вместо D-триггеров JK-триггеры. Общая процедура реализации определяется ответом задачи A.35.

A.37. В торговом автомате, описанном в разделе A.13.4, выдача товара ассоциировалась со значением на двоичном выходе z. Сдачу автомат не возвращал. Усовершенствуйте эту модель автомата так, чтобы он правильно возвращал сдачу. Предполагается, что он может получать монеты достоинством 10 и 25 центов в следующей последовательности: 10–10–10, 10–25, 25–10, 25–25. После того как в автомат будет опущена последняя монета, он должен вернуть соответственно 0, 5, 5 или 20 центов. Добавьте в него два новых двоичных выхода, z_2 и z_3, для представления трех различных выходных сигналов.

а) Составьте таблицу состояний, включающую новые выходы.

б) Составьте логические выражения для новых выходов z_2 и z_3.

в) Имеются ли в новой таблице эквивалентные состояния?

A.38. Конечный автомат может использоваться для определения факта вхождения заданных последовательностей значений в последовательность двоичных значений, подаваемых на вход автомата. Такой автомат называется распознающим конечным автоматом. Предположим, что в ответ на вторую единицу в каждой последовательности 011, поступившей на вход автомата, он выдает 1 на выходе.

а) Нарисуйте диаграмму состояний такого автомата.

б) Обозначьте состояния автомата, составьте таблицу состояний и нарисуйте схему его реализации. Предполагается, что для решения задачи применяются D-триггеры.

в) Повторите задачу, сформулированную в упражнении A.38, *а*, для автомата, распознающего входные последовательности 011 и 010, включая также случаи их пересечения. Например, для входной последовательности 110101011... должна генерироваться выходная последовательность 000010101... .

Приложение Б

Система команд процессора ARM

В этом приложении приведено краткое описание версии v3 архитектуры системы команд ARM, о которой рассказывается в первой части главы 3. Кроме того, в ней, как вы помните, перечислены изменения, внесенные в более поздние версии этой системы команд. Структура регистров ARM показана на рис. 3.1. На рис. 3.2 приведен общий формат команд. В данном приложении разные типы команд ARM описаны более подробно. Все они кодируются 32-разрядным словом. Память адресуется побайтово, при помощи 32-разрядных адресов. Поддерживаются два размера операндов: слово (32 бита) и байт (8 бит). Байтовый операнд занимает 8 бит регистра процессора. Когда он загружается в регистр, старшие 3 байта данного регистра очищаются путем установления их значений в 0.

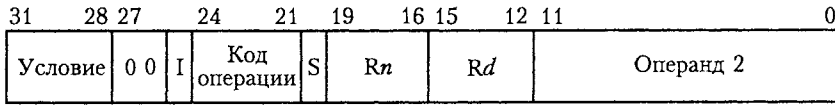
Б.1. Кодирование команд

На рис. Б.1 показано, как кодируется каждый из пяти типов команд ARM. Тип команды определяется битовым кодом, начинающимся с позиции b_{27}. Команды умножения, представленные на рис. Б.1, *б*, отличаются от группы команд, выполняющих другие арифметические и логические операции, которые перечислены на рис. Б.1, *а*. Это отличие состоит в следующем. Когда в указанной группе команд бит I, а также бит b_7 или b_4 равны 0, то, как и в случае команды умножения, значением обоих битов является 1. Обратите внимание, что поля R*n* и R*d* в команде умножения поменялись местами.

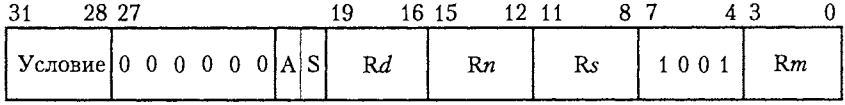
В следующих разделах мы более подробно поговорим о кодировании команд всех типов и приведем для каждого из них соответствующие примеры. Вы также получите необходимую информацию об имеющихся в архитектуре ARM дополнительных командах, связанных с выполнением операций сопроцессора.

Условное выполнение команд

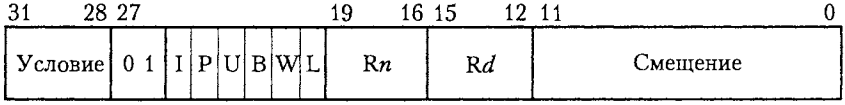
Коды условий для команд условного перехода приведены в табл. Б.1. Мнемоническое обозначение нужного условия добавляется к мнемоническому обозначению кода команды в виде суффикса. Условие AL определяет, что команда выполняется независимо от состояния флагов кодов условий. Если в программе на языке ассемблера суффикс условия не указан, по умолчанию считается заданным условие AL. Например, команды ADD (сложение) и B (переход) выполняются всегда, а команды ADDEQ и BEQ — только при условии Z = 1. Условный переход часто производится после выполнения команды сравнения. Учитывайте это при просмотре столбца названий возможных условий в табл. Б.1.



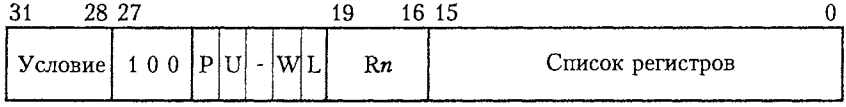
а



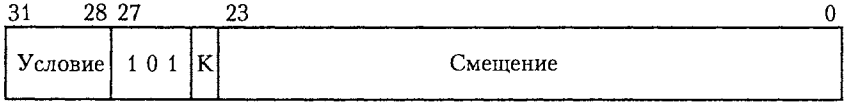
б



в



г



д

I — непосредственная адресация
P — преиндексация/постиндексация
W — обратная запись
S — установка флагов кодов условий
U — вверх/вниз
L — загрузка/сохранение
A — умножение с суммированием
B — байт/слово
K — условие связывания

Рис. Б.1. Форматы команд ARM: арифметика, логика, сравнение, проверка и пересылка (*а*); умножение и умножение с суммированием (*б*); пересылка в память и из памяти одного слова или байта (*в*); пересылка в память и из памяти нескольких слов (*г*); переход и переход со связыванием (*д*)

Таблица Б.1. Кодирование поля условия в командах ARM

Поле условия $b_{31}...b_{28}$	Суффикс условия	Название	Проверка кода условия
0 0 0 0	EQ	Равно (нуль)	Z = 1
0 0 0 1	NE	Не равно (не нуль)	Z = 0
0 0 1 0	CS/HS	Установка переноса/беззнаковое старше или равно	C = 1
0 0 1 1	CC/LO	Очистка переноса/беззнаковое младше	C = 0
0 1 0 0	MI	Минус (отрицательно)	N = 1
0 1 0 1	PL	Плюс (положительно или нуль)	N = 0
0 1 1 0	VS	Переполнение	V = 1
0 1 1 1	VC	Нет переполнения	V = 0
1 0 0 0	HI	Беззнаковое старше	C ∨ Z = 0
1 0 0 1	LS	Беззнаковое младше или равно	C ∨ Z = 1
1 0 1 0	GE	Со знаком больше или равно	N ⊕ V = 0
1 0 1 1	LT	Со знаком меньше	N ⊕ V = 1
1 1 0 0	GT	Со знаком больше	Z ∨ (N ⊕ V) = 0
1 1 0 1	LE	Со знаком меньше или равно	Z ∨ (N ⊕ V) = 1
1 1 1 0	AL	Всегда	
1 1 1 1		Не используется	

Б.1.1. Арифметические и логические команды

Арифметические и логические операции, а также команды сравнения, проверки и пересылки выполняются командами, формат которых показан на рис. Б.2. Первый операнд содержится в регистре R*n*, второй — в регистре R*m* или является беззнаковым непосредственно заданным 8-битовым значением, на что указывает бит I. Результат операции, определяемой 4-битовым кодом операции, помещается в регистр R*d*. Если бит S равен 1, результат влияет на флаги кодов условий; в противном случае (S = 0) флаги кодов условий не устанавливаются.

Общий синтаксис языка ассемблера для этих команд таков:

OP{Условие}{S} R*d*,R*n*,Операнд2

Например, если второй операнд содержится в регистре (I = 0), команда

ADD R0,R1,R2

выполняется условно и осуществляет операцию

[R0] ← [R1] + [R2]

не меняя флагов кодов условий. В том случае, если задан код операции ADDS, флаги кодов условий устанавливаются в соответствии с результатами операции.

Но если последняя команда должна быть выполнена при условии «равно» (EQ), код операции записывается как ADDEQS.

Если второй операнд задан непосредственно (I = 1), применяется выражение #константа. Так, команда

ADD R0,R1,#17

выполняет операцию

[R0] ← [R1] + 17

Перед использованием в операции непосредственно заданное значение дополняется нулями до 32 разрядов.

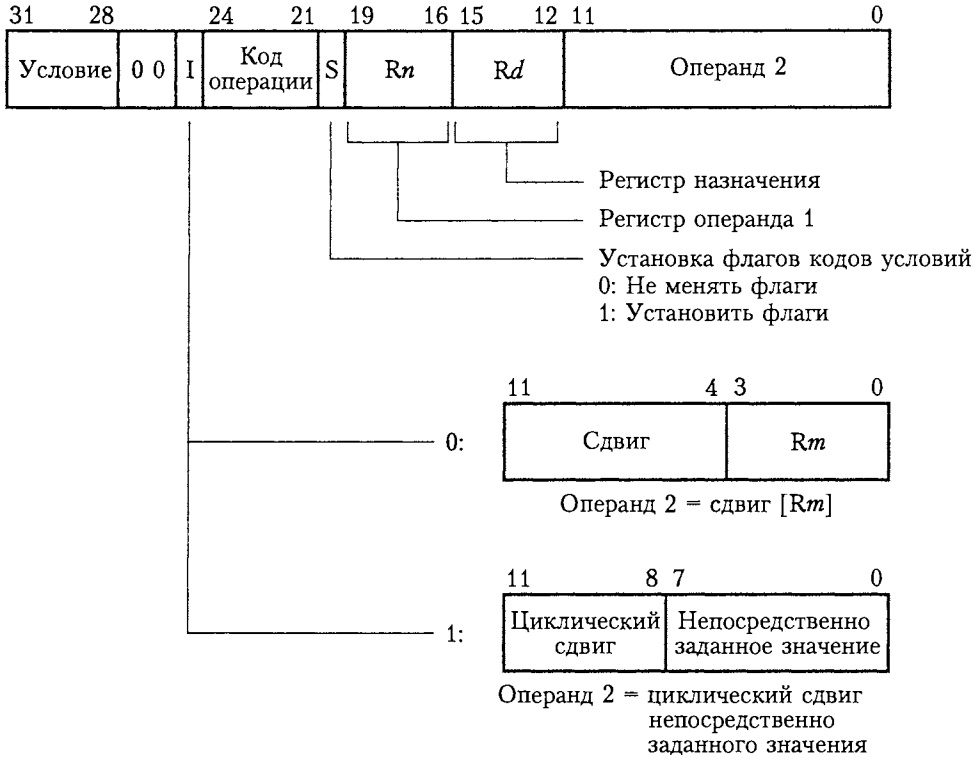


Рис. Б.2. Формат арифметических команд, логических команд, команд сравнения, проверки и пересылки процессоров ARM

Сдвиг операнда 2

Если операнд 2 содержится в регистре (I = 0), перед использованием он может быть сдвинут, как показано на рис. Б.3. Сдвиг задается в разрядах b_{11-4}. Если $b_4 = 0$, сдвиг в диапазоне от 0 до 31 задается как 5-битовое число без знака в разрядах b_{11-7}. Тип сдвига определяется битами b_6 и b_5. Флаг кода условия C, используемый в четырех операциях сдвига, показан на рис. 2.30 и 2.32. Когда расстояние сдвига не равно нулю, операция циклического сдвига вправо (ROR) выполняется

без бита C. Если же величина сдвига равна 0, это значит, что должен быть выполнен циклический сдвиг на один бит, включая бит C, как показано на рис. 2.32, *г*. На языке ассемблера данная операция обозначается как RRX (Rotate Right Extended), а величина сдвига не задается. Вот пример команды, в которой данный параметр указывается прямо в команде:

```
ADD  R0,R1,R2, LSL #4
```

Данная команда сдвигает операнд в регистре R2 на 4 разряда влево (умножая его тем самым на 16), а после этого добавляет результат к содержимому регистра R1. Если $b_4 = 1$, величина сдвига задается в младших пяти битах регистра R*s*, как показано на рис. Б.3. Так, команда

```
ADD  R0,R1,R2, LSR R3
```

сдвигает содержимое операнда R2 вправо на количество позиций, заданное в регистре R3.

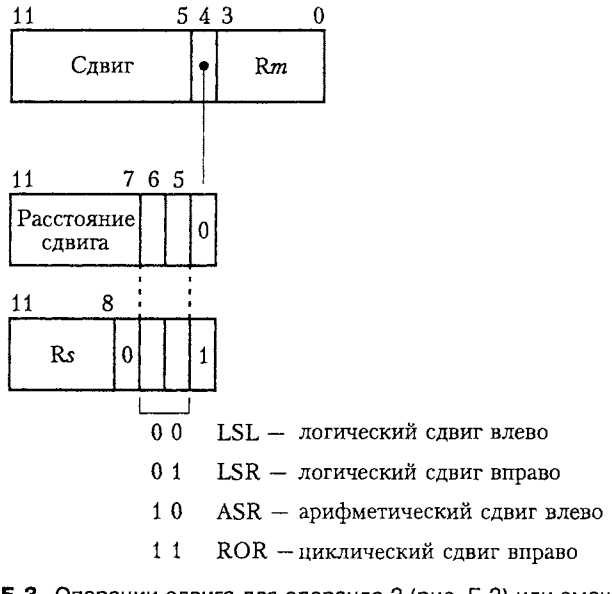


Рис. Б.3. Операции сдвига для операнда 2 (рис. Б.2) или смещения адреса, содержащегося в регистре R*m* (рис. Б.5)

Если операнд 2 задан непосредственно в команде (I = 0), он может быть циклически сдвинут вправо, как указано на рис. Б.2. Эта опция позволяет генерировать большое количество 32-разрядных значений путем сдвига беззнакового непосредственно заданного 8-разрядного значения. Количество позиций сдвига составляет $2n$, где n — это 4-битовое число, содержащееся в битах b_{11-8}. Таким образом, величина смещения соответствует четному количеству битов из диапазона от 0 до 30. Эта функция системы команд ARM отчасти компенсирует невозможность задавать 32-разрядные значения непосредственно в командах. Но не все 32-разрядные значения можно сгенерировать подобным образом. Для формирования 32-разрядных значений, которые нельзя задать при помощи сдвига одного

8-разрядного значения, может использоваться короткая последовательность команд, включающая операции циклического сдвига и логического ИЛИ.

Полный набор команд, включающий всего 16 арифметических и логических команд, представлен в двух таблицах — Б.2 и Б.3.

Таблица Б.2. Арифметические команды ARM

Мнемоническое обозначение (название)	Код операции $b_{24}...b_{21}$	Выполняемая операция	Флаги кодов условий, устанавливаемые, если S = 1			
			N	Z	V	C
ADD (Сложение)	0 1 0 0	Rd ← [Rn] + Опер2	x	x	x	x
ADC (Сложение с переносом)	0 1 0 1	Rd ← [Rn] + Опер2 + [C]	x	x	x	x
SUB (Вычитание)	0 0 1 0	Rd ← [Rn] − Опер2	x	x	x	x
SBC (Вычитание с переносом)	0 1 1 0	Rd ← [Rn] − Опер2 + [C] −1	x	x	x	x
RSB (Обратное вычитание)	0 0 1 1	Rd ← Опер2 − [Rn]	x	x	x	x
RSC (Обратное вычитание с переносом)	0 1 1 1	*Rd* ← Опер2 − [Rn] + [C] − 1	x	x	x	x
MUL (Умножение)	Представлен на рис. Б.4	Rd ← [Rm] × [Rs]	x	x		
MLA (Умножение со сложением)	Представлен на рис. Б.4	Rd ← [Rm] × [Rs] + [Rn]	x	x		

Таблица Б.3. Логические команды ARM, команды сравнения, проверки и пересылки

Мнемоническое обозначение (название)	Код операции $b_{24}...b_{21}$	Выполняемая операция	Флаги кодов условий, устанавливаемые, если S = 1			
			N	Z	V	C
AND (Логическое И)	0 0 0 0	*Rd* ← [*Rn*] ∧ Опер2	x	x		x
ORR (Логическое ИЛИ)	1 1 0 0	*Rd* ← [*Rn*] ∨ Опер2	x	x		x
EOR (Исключающее ИЛИ)	0 0 0 1	*Rd* ← [*Rn*] ⊕ Опер2	x	x		x

Таблица Б.3 (*продолжение*)

Мнемоническое обозначение (название)	Код операции $b_{24}...b_{21}$	Выполняемая операция	Флаги кодов условий, устанавливаемые, если S = 1			
			N	Z	V	C
BIC (Очистка битов)	1 1 1 0	R*d* ← [R*n*] ∧ ¬Опер2	x	x		x
CMP (Сравнение)	1 0 1 0	[R*n*] − Опер2	x	x	x	x
CMN (Отрицательное сравнение)	1 0 1 1	[R*n*] − Опер2	x	x	x	x
TST (Проверка битов)	1 0 0 0	[R*n*] ∧ Опер2	x	x		x
TEQ (Проверка на равенство)	1 0 0 1	[R*n*] ⊕ Опер2	x	x		x
MOV (Пересылка)	1 1 0 1	R*d* ← Опер2	x	x		x
MVN (Пересылка дополнения)	1 1 1 1	R*d* ← ¬Опер2	x	x		x

К числу таких команд относятся две команды умножения, а также шесть команд сложения и вычитания. Команды сложения с переносом и вычитания с переносом нужны для обработки операндов, состоящих из нескольких слов. Так как сдвигать можно только операнд 2, в наборе предусмотрены еще и две команды обратного вычитания, позволяющие сдвигать первый операнд операции вычитания. Когда непосредственно заданное значение используется в командах сложения и вычитания в качестве операнда 2, оно может быть только положительным. Но в языке ассемблера допускаются и команды типа

```
ADD  R0,R1,#-5
```

и

```
SUB  R0,R1,#-7
```

Они просто ассемблируются как

```
SUB  R0,R1,#5
```

и

```
ADD  R0,R1,#7
```

Как видите, в дополнение к четырем логическим командам AND, ORR, EOR и BIC имеется команда пересылки дополнения (MVN), выполняющая логическую операцию НЕ. Команды сравнения и проверки всегда воздействуют на флаги кодов условий.

В вашем распоряжении также есть две команды Move, выполняющие пересылку операнда 2 или его поразрядного дополнения в регистр назначения. Операнд 2 может содержаться в регистре или задаваться непосредственно в команде. Наряду с пересылкой данных между регистрами эти две команды используются и для загрузки констант в регистры. Команду MVN можно применить и для загрузки отрицательных чисел в формате дополнения до двух. Для этого в программе на языке ассемблера может быть задействована команда

```
MOV  R0,#-10
```

которая ассемблируется как

```
MVN  R0,#9
```

Поразрядным дополнением десятичного числа 9 (0...01001) является двоичное значение 1...10110 — это и есть представление числа −10 в формате дополнения до двух.

Команды умножения

Формат двух команд умножения и выполняемые ими операции представлены соответственно на рис. Б.4 и в табл. Б.2. Ни один из операндов этих команд не может быть сдвинут. Результатом операции умножения является 32-разрядное значение одинарной точности.

MUL: $Rd \leftarrow [Rm] \times [Rs]$
MLA: $Rd \leftarrow [Rm] \times [Rs] + [Rn]$

Рис. Б.4. Команды ARM умножения и умножения со сложением

Б.1.2. Команды загрузки данных из памяти и их сохранения в памяти

Формат двух команд, используемых для доступа к памяти, показан на рис. Б.5, а выполняемые ими операции перечислены в табл. Б.4. Бит L в позиции b_{20} равен 1 для команды загрузки (LDR) и 0 для команды сохранения (STR). Бит B в позиции b_{22} равен 1 для команды байтового операнда и 0 для 32-разрядного слова. Байтовый операнд располагается в младших разрядах регистра R*d*. Исполнительный адрес операнда в памяти определяется путем прибавления (U = 1) или вычитания

(U = 0) величины смещения, заданной в поле «Смещение», к содержимому или из содержимого регистра R*n*. Как следует из рис. Б.5 и табл. Б.3, биты P и W определяют операции преиндексации или постиндексации и обратной записи. Обратите внимание, что содержимое регистра R*m* может быть сдвинуто одним из двух возможных методов сдвига.

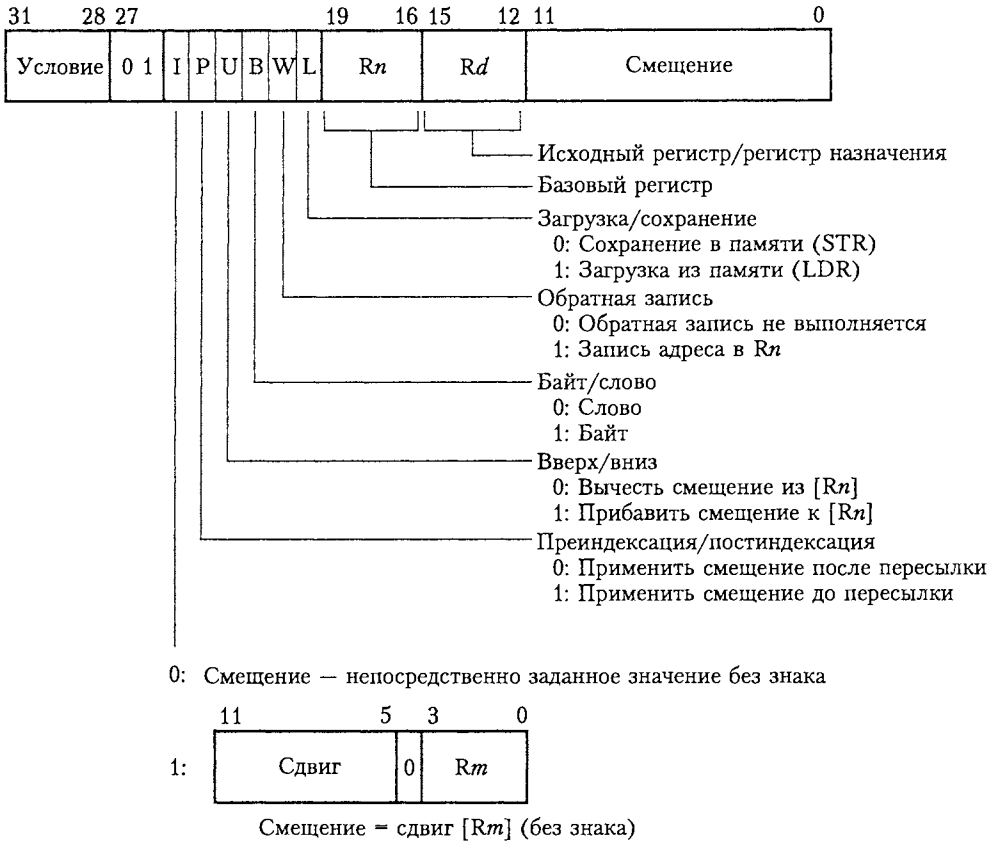


Рис. Б.5. Команды загрузки и сохранения процессоров ARM

Таблица Б.4. Команды ARM для пересылки в память и из памяти одного слова или байта

Мнемоническое обозначение (название)	Биты команды B L	Выполняемая операция
LDR (Загрузка слова)	0 1	R*d* ← [EA]
LDRB (Загрузка байта)	1 1	R*d* ← [EA]
STR (Сохранение слова)	0 0	EA ← [Rd]
STRB (Сохранение байта)	1 0	EA ← [Rd]

Ниже дано несколько примеров операций доступа к памяти. Для команды

```
LDR  R0,[R1,#100]
```

выполняется операция

$$R0 \leftarrow [[R1] + 100]$$

Биты этой команды установлены следующим образом: I = 0, P = 1, U = 1, B = 0, W = 0, L = 1. Смещение находится в диапазоне ±4095. Для команды

```
LDR  R0,[R1,R2]
```

выполняется операция

$$R0 \leftarrow [[R1] + [R2]]$$

В этой команде бит I = 1, а значения остальных битов те же, что и в предыдущей команде.

Когда информация о смещении содержится в регистре, перед изменением содержимого базового регистра R*n* значение смещения может быть сдвинуто. Сдвиг задается 5-битовым значением с использованием непосредственной адресации, как показано на рис. Б.3. Например, команда

```
LDR  R0,[R1,−R2, LSL #4]!
```

выполняет операцию

$$R0 \leftarrow [[R1] - 16 \times [R2]]$$

а исполнительный адрес записывается обратно в R1. Биты этой команды установлены так: I = 1, P = 1, U = 0, B = 0, W = 1, L = 1.

Если в качестве базового регистра задан счетчик команд R15, то, как видно из табл. 3.1, применяется относительный режим адресации. В этом случае для формирования исполнительного адреса операнда используется преиндексация без обратной записи с непосредственно заданным смещением. В случае применения абсолютной адресации ассемблер во время выполнения команды вычисляет смещение относительно обновленного содержимого счетчика команд. Например, если команда

```
LDR  R0,PARAMETER
```

хранится по адресу 1000, а метка PARAMETER представляет адрес 1100, то ассемблер генерирует команду

```
LDR  R0,[R15,#92]
```

К тому моменту, когда величина смещения прибавляется к содержимому счетчика команд, последний уже обновлен и содержит значение 1008, поэтому для получения правильного исполнительного адреса смещение должно быть равным 92: 1000 = 1008 + 92.

Б.1.3. Команды блочной загрузки и сохранения

На рис. Б.6 показано, как кодируются команды, выполняющие пересылку данных между блоком последовательных слов памяти и заданным подмножеством из 16 регистров процессора. Для загрузки операндов из памяти в регистры используется код операции LDM (Load multiple — загрузка групповая), а для сохранения операндов в памяти — код операции STM (Store multiple — запись групповая). Для операции загрузки бит L устанавливается в 1, а для операции сохранения — в 0. Участвующие в операции регистры задаются единицами в 16-разрядном поле списка регистров, в разрядах b_{15-0}. Адрес начала блока слов в памяти определяется содержимым базового регистра R*n*. Если бит U равен 1, блок продолжается в направлении увеличения адресов, а если он равен 0 — в направлении их уменьшения. Преиндексация или постиндексация регистра R*n* определяется битом P. Значение индекса всегда равно 4, поскольку операнды являются последовательными 4-байтовыми словами. Последний адрес, сгенерированный при выполнении блочной пересылки, записывается в регистр R*n*, если бит W равен 1; в противном случае (W = 0) в регистре R*n* остается начальный адрес. Независимо от того, с какой стороны от начального адреса находится блок, регистр с наименьшим номером всегда ассоциируется с младшим адресом в блоке. В табл. Б.5 показаны мнемонические обозначения всех возможных состояний битов L, P и U. Состояние битов P и U обозначается суффиксами, прибавляемыми к кодам операций LDM и STM. Так, значения P = 0 и U = 1 в первой строке табл. Б.5 обозначаются суффиксом IA (Increment After — увеличение адресов и постиндексация), указывающим, что после каждой пересылки содержимое базового регистра R*n* увеличивается на 4. Альтернативные мнемонические обозначения, приведенные в табл. Б.5, рассматриваются далее в этом разделе.

Основной целью применения блочной пересылки является сохранение регистров в стеке при входе в подпрограммы и их восстановление при выходе из таковых. Если регистр R13 используется как указатель стека, а в регистре R14 (регистре связи) хранится адрес возврата, то команда

```
STMDB  R13!,{R0-R3,R14}
```

описанная в последней строке табл. Б.5, проталкивает в стек содержимое регистров R0–R3, а также регистра R14. Стек растет в направлении уменьшения адресов памяти, поэтому содержимое регистра R0 помещается в него последним, по наименьшему адресу. Соответствующая команда

```
LDMIA  R13!,{R0-R3,R15}
```

описанная в первой строке табл. Б.5, выталкивает из стека сохраненное содержимое регистров R0–R3, помещая его обратно в эти же регистры, а сохраненное значение регистра R14 (адрес возврата) выталкивает в регистр R15, являющийся счетчиком команд. Таким образом выполняется операция возврата из подпрограммы. Содержимое старшего адреса пересылается в регистр R15 последним. Суффиксы DB и IA в этих двух кодах операций означают decrement before и increment after, в зависимости от того, какие операции выполняются с содержимым базового регистра. Альтернативными мнемоническими обозначениями тех

же команд являются STMFD и LDMFD. Суффикс FD происходит от full descending. Это обозначение отражает тот факт, что стек растет в направлении уменьшения адресов (descending) и начальным содержимым базового регистра R13 является адрес текущего верхнего элемента стека (full). Для стека, который растет в направлении увеличения адресов и указатель которого указывает на пустое место над текущим верхним элементом, применяются обозначение empty ascending, а следовательно, суффикс EA. Об использовании команд LDM и STM для входа в подпрограммы и выхода из таковых рассказывается в главе 4.

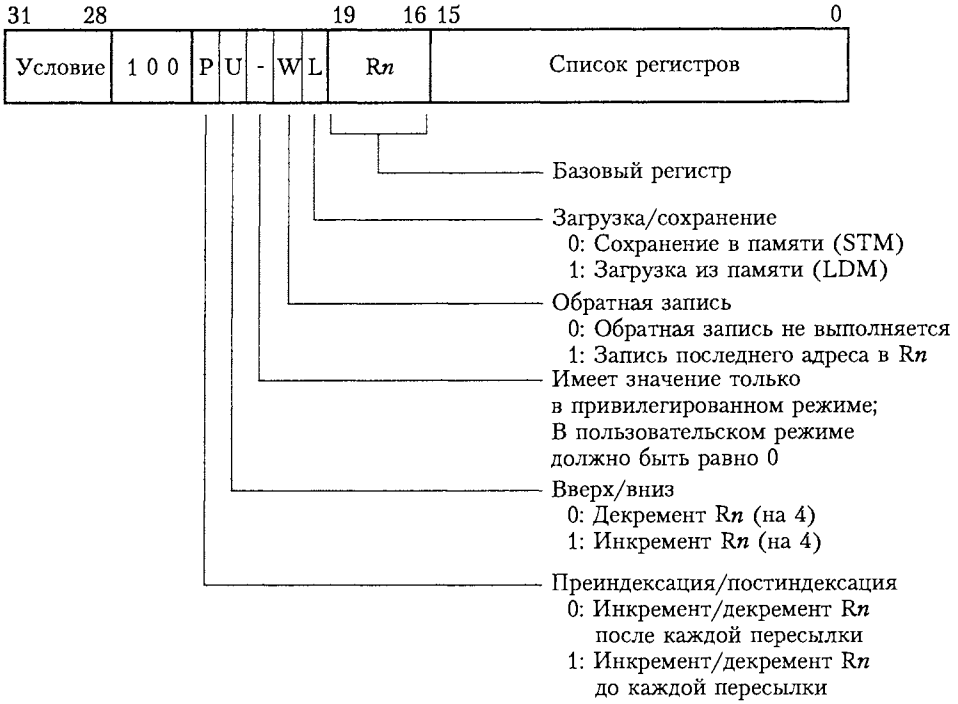


Рис. Б.6. Команды блочной пересылки процессоров ARM

Таблица Б.5. Команды ARM для пересылки в память и из памяти нескольких слов

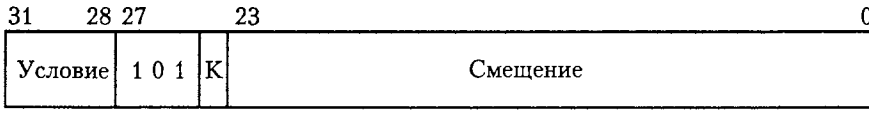
Мнемоническое обозначение	Биты команды			Выполняемая операция
	P	U	L	
LDMIA/LDMFD	0	1	1	$R_{мл}, ..., R_{ст} \leftarrow [[Rn]], [[Rn] + 4],...$
LDMIB/LDMED	1	1	1	$R_{мл}, .., R_{ст} \leftarrow [[Rn] + 4], [[Rn] + 8],...$
LDMDA/LDMFA	0	0	1	$R_{мл}, ..., R_{ст} \leftarrow [[Rn]], [[Rn] - 4],...$
LDMDB/LDMEA	1	0	1	$R_{мл}, ..., R_{ст} \leftarrow [[Rn] - 4], [[Rn] - 8],...$
STMIA/STMEA	0	1	0	$[Rn], [Rn] + 4,... \leftarrow [R_{мл}], ..., [R_{ст}]$

Таблица Б.5 (*продолжение*)

Мнемоническое обозначение	Биты команды P U L	Выполняемая операция
STMIB/STMFA	1 1 0	[Rn] + 4, [Rn] + 8,... ← [$R_{мл}$], ..., [$R_{ст}$]
STMDA/STMED	0 0 0	[Rn], [Rn] − 4,... ← [$R_{ст}$], ..., [$R_{мл}$]
STMDB/STMFD	1 0 0	[Rn] − 4, [Rn] − 8,... ← [$R_{ст}$], ..., [$R_{мл}$]

Б.1.4. Команды перехода и перехода со связыванием

На рис. Б.7 проиллюстрирован принцип кодирования команд перехода (B) и перехода со связыванием (BL). Величина смещения представляется 24-разрядным числом со знаком. Оно сдвигается на две битовые позиции влево (все целевые адреса перехода выравниваются по адресам слов), затем его знак расширяется до 32 бит и полученное значение прибавляется к обновленному содержимому регистра PC, в результате чего получается целевой адрес перехода. Обновленный регистр PC указывает на команду, которая находится на два слова (то есть на 8 байт) впереди команды перехода.



K = 0: Переход (B)

K = 1: Переход со связыванием (BL); сохранение адреса возврата в регистре R14

Рис. Б.7. Команды перехода и перехода со связыванием

Язык ассемблера позволяет задать в команде целевой адрес перехода. Так, команда

BEQ ROUTINE

при условии Z = 1 выполняет переход по адресу ROUTINE. Если команда перехода располагается по адресу 2000, а метка ROUTINE — по адресу 3000, вычисляемое значение смещения должно быть задано в команде как 248. Реальное расстояние от обновленного содержимого счетчика команд (2008) до целевого адреса составляет 992. Это значение является числом 248, сдвинутым влево на 2 разряда (то есть умноженным на 4). Другими словами, целевой адрес перехода вычисляется так: 2008 + 992 = 3000.

Команда перехода со связыванием (BL) предназначена для вызова подпрограмм. Еще до перехода к подпрограмме адрес команды, непосредственно следующей за командой BL (адрес возврата), сохраняется в регистре R14, который используется в качестве регистра связи. Как осуществляется возврат из подпрограммы, описано в разделе 3.6.

Б.1.5. Команды управления компьютером

Программные прерывания

Когда выполнение пользовательской программы завершается, команда программного прерывания SWI передает управление программе-супервизору, являющейся частью операционной системы. Мы не упоминали о команде SWI в примерах программ для процессора ARM, представленным в главе 3, хотя она должна была там присутствовать. Так, в программе, приведенной на рис. 3.8, она должна следовать за командой STR. У команды SWI имеется еще одно назначение: она используется для передачи управления подпрограммам операционной системы, работающим в режиме супервизора, при выполнении пользовательскими программами операций ввода-вывода (см. главу 4).

Формат команды SWI на языке ассемблера, а также выполняемые ею операции указаны в табл. Б.6. Код операции команды, 1111, располагается в битовом поле b_{27-24}. Подобно всем остальным командам ARM, данная команда может выполняться условно. В младших 24 разрядах содержится непосредственно заданный операнд, игнорируемый в ходе выполнения команды. Пользовательская программа может применить это поле для передачи операционной системе параметра, определяющего запрошенный сервис, скажем, операцию ввода-вывода.

Таблица Б.6. Команды ARM для пересылки содержимого регистра состояния, обработки программных прерываний и используемые при необходимости поменять данные местами

Мнемоническое обозначение (название)	Формат команды	Выполняемая операция
MRS (Копирование регистра состояния)	Пользовательский режим: MRS R*d*,CPSR	Rd ← [CPSR]
	Привилегированный режим: MRS R*d*,CPSR MRS R*d*,SPSR	R*d* ← [CPSR] R*d* ← [SPSR_*режим*]
MSR (Запись в регистр состояния)	Пользовательский режим: MSR CPSR,R*m* MSR CPSR,непоср32	$CPSR_{31-28}$ ← $[Rm]_{31-28}$ $CPSR_{31-28}$ ← $непоср32_{31-28}$
	Привилегированный режим: MSR CPSR,R*m* MSR CPSR_флаг,R*m* MSR CPSR_флаг,непоср32 MSR SPSR,R*m* MSR SPSR_флаг,R*m* MSR SPSR_флаг,непоср32	$CPSR_{31-28}$ ← [R*m*] $CPSR_{31-28}$ ← $непоср32_{31-28}$ SPSR_*режим* ← [R*m*] $SPSR\_режим_{31-28}$ ← $[Rm]_{31-28}$ $SPSR\_режим_{31-28}$ ← $непоср32_{31-28}$
SWI (Программное прерывание)	SWI непоср24	R14_svc ← обновленный[PC]; SPSR_svc ← [CPSR]; PC ← 0x08
SWP (Обмен)	SWP R*d*, R*m*,[R*n*]	R*d* ← [[R*n*]]; [R*n*] ← [R*m*]

Пересылка регистра состояния процессора

В архитектуре ARM имеются команды для управления регистром текущего состояния процессора CPSR и сохраненным регистром текущего состояния процессора SPSR_*режим* (рис. 3.1 и 4.12), предназначенные главным образом для использования в программах привилегированного режима. Допускается ограниченное применение этих команд и в пользовательских программах. Когда внешнее прерывание приостанавливает выполнение пользовательской программы, текущее содержимое регистра CPSR автоматически сохраняется в регистре SPSR_*режим*, на то время, пока программа привилегированного режима будет обрабатывать прерывание. Такие программы должны манипулировать содержимым регистра состояния (этот процесс рассматривался в главе 4). Для чтения и записи содержимого регистров CPSR и SPSR_*режим* используются описанные в табл. Б.6 команды MRS и MSR. Чтение регистров состояния разрешается любым программам — как пользовательским, так и привилегированным. Программы привилегированного режима могут записывать все 32 бита регистров CPSR и SPSR_*режим* или выборочно устанавливать отдельные поля флагов кодов условий. Исходным операндом операций записи является либо содержимое регистра общего назначения, либо 32-разрядное значение, непосредственно заданное в команде и обозначенное в табл. Б.6 как *непоср32*. В операции используются только старшие 4 бита непосредственно заданного операнда, поэтому его всегда можно сгенерировать путем циклического сдвига указанного в команде короткого 8-битового значения поля.

Команды MRS и MSR преобразуются в тот же машинный формат, что и арифметические и логические команды (рис. Б.1). Команды CMP, CMN, TST и TEQ, указанные в табл. Б.3, обязательно устанавливают флаги кодов условий. Поэтому значение бита S в них всегда равно 1. Когда этот бит установлен в 0, те же четыре кода операций представляют команды MRS и MSR, оперирующие регистрами CPSR и SPSR_*режим*. Остальные разряды команд предназначены для задания режима полной или частичной записи, а также для определения исходного операнда команды MSR с использованием регистровой или непосредственной адресации.

Обмен данными между регистрами и памятью

В архитектуре ARM имеется команда, которая считывает содержимое памяти в один регистр и записывает туда же содержимое другого регистра. Речь идет о команде SWP (табл. Б.6). Она выполняется и в пользовательском, и в привилегированном режимах. Основным назначением команды является выполнение операций с переменными блокировки для координирования процесса обработки данных, расположенных в памяти и совместно используемых несколькими программами в мультипроцессорных конфигурациях (см. главу 12). Мы назвали команду SWP «непрерываемой» потому, что между выполняемыми ею операциями чтения и записи ни одному другому процессору доступ к памяти не предоставляется. В регистрах R*m* и R*d* может быть указан один и тот же регистр, в результате чего будет выполнен обмен значениями между этим регистром и операндом в памяти.

Б.2. Другие команды ARM

В этом разделе коротко описываются команды сопроцессора и команды, добавленные в версии v4 и v5 архитектуры системы команд ARM.

Б.2.1. Команды сопроцессора

Аппаратные блоки для выполнения операций, не включенных в систему команд ARM, называются сопроцессорами. Примером сопроцессора является аппаратный блок для выполнения операций над числами с плавающей запятой. Другими примерами могут служить блоки для выполнения специфической для конкретного устройства цифровой обработки сигналов или видеоданных, которые могут применяться во встроенных системах. Если спецификация процессора ARM определена в программно-синтезируемой форме (см. главы 9 и 11), программный модуль сопроцессора может быть интегрирован в программное описание процессора и использоваться для реализации обоих, процессора и сопроцессора, на одной микросхеме. Возможность программировать комбинируемые блоки обеспечивается включением в систему команд ARM шаблонов команд для передачи сопроцессору информации об операциях, которые он должен выполнять, пересылки данных между регистрами сопроцессора и памятью, а также для пересылки данных между регистрами сопроцессора и регистрами ARM.

Б.2.2. Команды версий v4 и v5

В две версии системы команд ARM, следующие за версией v3, были добавлены команды для доступа к памяти и дополнительные команды, выполняющие операцию умножения. В версиях v4 и v5 имеются команды загрузки и сохранения, пересылающие между памятью и регистрами процессора байты со знаком, а также 16-битовые полуслова со знаком и без знака. Процессоры ARM выполняют операции только над 32-битовыми операндами. Когда любая из команд версии v4 или v5 загружает в регистры процессора байт со знаком (или полуслово со знаком), его знак расширяется до 32 разрядов.

Кроме того, в версиях v4 и v5 ARM появились дополнительные формы команд MUL и MULA, имевшихся в версии v3 (рис. Б.4). Существуют версии этих команд со знаком и без знака, генерирующие 64-разрядные произведения.

Б.3. Программирование

На web-узле ARM по адресу www.arm.com/hr.ns4/html/SDT202u представлены средства разработки программного обеспечения, которые могут использоваться для ввода, редактирования, ассемблирования и эмулируемого запуска программ на языке ассемблера ARM. При необходимости ассемблировать программу, приведенную на рис. 3.8, директивы AREA должны быть заменены директивами

```
AREA    addloop,      CODE
AREA    addloopdata,  DATA
```

К тому же, как было сказано в разделе Б.1.5, после команды STR необходимо добавить команду программного прерывания в форме

```
SWI   0x123456
```

Приложение B

Система команд процессора Motorola 68000

Об основных характеристиках процессора Motorola 68000 речь шла во второй части главы 3, где, в частности, было рассказано о структуре регистров (рис. 3.18) и режимах адресации. Обратите внимание, что в табл. 3.2 синтаксис режимов адресации приведен для языка ассемблера. В настоящем приложении дано краткое описание системы команд процессора.

Адресация и коды операций

Общий формат адресных полей операнда показан в табл. B.1. Как видите, режим адресации и применяемые в команде регистры определяет 6-битовое поле. В тех режимах, для которых не требуется указывать конкретный регистр, все 6 бит используются для задания режима адресации. В табл. B.1 даны названия режимов адресации, применявшиеся в книге. Но в документации Motorola некоторые из них называются по-другому. А так как читателю, возможно, придется обращаться к документации производителя, в табл. B.2 приведены оба варианта терминов — используемые нами и предлагаемые компанией. Терминология Motorola очень информативна, но для нашей книги несколько громоздка.

В данном приложении команды процессора 68000 представлены в форме таблицы. Чтобы таблица получилась не слишком большой, мы в некоторых случаях использовали аббревиатуры. Эти аббревиатуры и их значения приведены в табл. B.3. Обратите внимание, что в поле кода операции каждый бит обозначен одной буквой. В табл. B.4 приведен полный список команд. Режимы адресации, которые можно использовать в этих командах, перечислены в виде матрицы. Разрешенные режимы адресации каждого исходного (результирующего) операнда отмечены символом «x». Так, в команде AND, если исходный операнд является регистром данных, для операнда назначения допускаются режимы (An), (An)+, −(An), d(An), d(An,Xi), Abs.W и Abs.L. Если же операнд назначения является регистром данных, то исходный операнд может быть задан в любом из 11 режимов.

В столбце кода операции показан реальный набор битов первого 16-битового слова команды. Если исходные данные команды задаются непосредственно, они занимают второе слово в случае 8- или 16-битового операнда либо второе и третье слова в случае 32-битового операнда. Для индексных и относительных режимов адресации необходимое значение индекса (то есть смещение) задается в слове, следующем за кодом операции.

Команды сдвига и циклического сдвига могут определять количество разрядов, на которое должен быть сдвинут исходный операнд. Эта информация задается

в регистре данных или прямо в команде как 3-битовое значение в поле кода операции. При использовании операнда, хранящегося в памяти, количество разрядов сдвига всегда равно 1.

Команды перехода перечислены в табл. В.5. Смещение перехода представляется числом со знаком в формате дополнения до двух, определяющим относительное расстояние в байтах. Возможные значения суффикса кода условия для команд условного перехода, а также для команд Scc приведены в табл. В.6. Здесь же указаны и условия, проверяемые перед выполнением перехода.

В табл. В.4 и В.5 перечислены операции, выполняемые каждой из команд. Однако по некоторым командам требуются дополнительные пояснения. В столбце мнемонических обозначений такие команды помечены звездочками. Они обсуждаются в следующих разделах.

Команды BCHG, BCLR, BSET и BTST

Все эти команды проверяют значение заданного бита операнда назначения. Номер проверяемого бита (bit#) указывается либо в регистре данных, либо прямо в команде. Проверка выполняется путем записи дополнения проверяемого бита в позицию флага условия Z.

Команда MOVEM

Команда пересылает значения одного или нескольких регистров в память или из памяти, где они располагаются по последовательным адресам. Участвующие в пересылке регистры задаются во втором слове команды. Биты от 0 до 7 соответствуют регистрам D0–D7, а биты от 8 до 15 соответствуют регистрам A0–A7. Такая организация принята во всех режимах адресации, за исключением автодекрементного, для которого характерен обратный порядок регистров.

Команда MOVEP

Эта команда предназначена в первую очередь для пересылки данных между процессором 68000 и 8-разрядными периферийными устройствами. Данные пересылаются побайтово, и после каждой операции пересылки байта адрес памяти увеличивается на 2. Таким образом, если начальный адрес памяти четный, все пересылаемые байты имеют четные адреса. Для их пересылки используются старшие восемь линий шины данных. Аналогичным образом, если начальный адрес памяти является нечетным, все пересылаемые байты имеют нечетные адреса. Для их пересылки применяются младшие восемь линий шины данных. Причем сначала пересылается старший байт регистра данных, а затем — младший.

Процессор 68000 может работать в одном из двух основных режимов. В режиме супервизора обычно выполняются любые команды, однако в режиме пользователя реализация некоторых из них невозможна. Команды, используемые только в режиме супервизора, называются привилегированными. Вот их перечень:

- ✦ команды ANDI, EORI, ORI и MOVE, в которых операндом назначения является регистр состояния SR;
- ✦ команда MOVE, пересылающая содержимое указателя пользовательского стека в регистр адреса или из регистра адреса;
- ✦ команды RESET, RTE и STOP.

Надеемся, что изложенной в данном приложении информации вполне достаточно для того, чтобы читатель мог самостоятельно писать и отлаживать программы на языке ассемблера процессора 68000. Размер и структуру ассемблированных команд можно определить на основе кодов операции и адресных режимов. По причине ограниченного объема книги мы не приводим некоторые из временных характеристик (например, количество машинных тактов, необходимых для выполнения каждой из команд). Эта информация, а также более подробные сведения о системе команд процессора 68000 содержатся в документации производителя.

Таблица В.1. Кодирование поля адреса для процессора 68000

Адресация	Поле режима адресации	Поле регистра
Непосредственная, регистр данных	000	Номер регистра
Непосредственная, регистр адреса	001	Номер регистра
Косвенная регистровая	010	Номер регистра
Автоинкрементная	011	Номер регистра
Автодекрементная	100	Номер регистра
Базовая индексная	101	Номер регистра
Полная индексная	110	Номер регистра
Абсолютная короткая	111	000
Абсолютная длинная	111	001
Базовая относительная	111	010
Полная относительная	111	011
Непосредственная	111	100

Таблица В.2. Соответствие терминологий — используемой в книге и предлагаемой компанией Motorola

Термины книги	Термины Motorola
Автоинкрементная	Address register indirect with postincrement
Автодекрементная	Address register indirect with predecrement
Базовая индексная	Address register indirect with displacement
Полная индексная	Address register indirect with index
Базовая относительная	Program counter with displacement
Полная относительная	Program counter with index

Таблица В.3. Обозначения, используемые в табл. В.4

Обозначение	Значение
S	Исходный операнд
D	Операнд назначения
An	Адресный регистр n
Dn	Регистр данных n
Xn	Регистр адреса или данных, используемый в качестве индексного
PC	Счетчик команд
SP	Указатель стека
SR	Регистр состояния
CCR	Флаги кодов условий в SR
AAA	Номер регистра адреса
DDD	Номер регистра данных
rrr	Номер исходного регистра
RRR	Номер регистра назначения
eeeeee	Исполнительный адрес исходного операнда
EEEEEE	Исполнительный адрес операнда назначения
MMM	Режим задания исполнительного адреса назначения
CCCC	Спецификация проверки кода условия
P...P	Смещение
Q...Q	Быстрые непосредственно заданные данные
SS	Размер: 00 ≡ байт, 01 ≡ слово, 10 ≡ длинное слово (для большинства команд); 01 ≡ байт, 11 ≡ слово, 10 ≡ длинное слово (для команд MOVE и MOVEA)
VVVV	Номер вектора ловушки
u	Неопределенное состояние флага кода условия (без смысла)
d(An)	Базовая индексная адресация
d(An,Xi)	Полная индексная адресация
d(PC)	Базовая относительная адресация
d(PC,Xi)	Полная относительная адресация
immed	Непосредственно заданное значение
bit#	Номер бита
count	Количество разрядов, на которое производится сдвиг
disp	Смещение
B	Байт
W	Слово
L	Длинное слово

Таблица В.4. Набор команд процессора 68000

Мнемоническое обозначение (название)	Размер	Режим адресации	Режим адресации												
			Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR или CCR
ABCD (Сложение BCD)	B	s = Dn d =	x												
		s = −(An) d =					x								
ADD (Сложение)	B,W,L	s = Dn d =	x		x	x	x	x	x	x	x				
		d = Dn s =	x	x	x	x	x	x	x	x	x	x	x	x	
ADDA (Сложение адресов)	W	d = An s =	x	x	x	x	x	x	x	x	x	x	x	x	
	L	d = An s =	x	x	x	x	x	x	x	x	x	x	x	x	
ADDI (Сложение с непосредственно заданным операндом)	B,W,L	s = Immed d =	x		x		x	x	x	x	x	x	x		
ADDQ (Сложение быстрое)	B,W,L	s = Immed3 d =	x	x	x	x	x	x	x	x	x				
ADDX (Сложение расширенное)	B,W,L	s = Dn d =	x												
		s = −(An) d =					x								
AND (Логическое И)	B,W,L	s = Dn d =	x		x	x	x	x	x	x	x				
		d = Dn s =	x		x	x	x	x	x	x	x	x	x	x	
ANDI (Логическое И с непосредственно заданным операндом)	B,W,L	s = Immed d =	x		x	x	x	x	x	x	x				x
ASL (Арифметический сдвиг влево)	B,W,L	count = [Dn] d =	x												
		count = QQQ d =	x												
		count = 1 d =			x	x	x	x	x	x	x				
ASR (Арифметический сдвиг вправо)	B,W,L	count = [Dn] d =	x												
		count = QQQ d =	x												
		count = 1 d =			x	x	x	x	x	x	x				
BCHG* (Проверка бита и его изменение)	B	bit# = [Dn] d =			x	x	x	x	x	x	x				
		bit# = Immed d =			x	x	x	x	x	x	x				
	L	bit# = [Dn] d =	x												
		bit# = Immed d =	x												
BCLR* (Проверка бита и его очистка)	B	bit# = [Dn] d =			x	x	x	x	x	x	x				
		bit# = Immed d =			x	x	x	x	x	x	x				
	L	bit# = [Dn] d =	x												
		bit# = Immed d =	x												

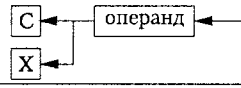
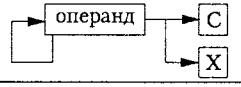
Код операции $b_{15}...b_0$	Выполняемая операция	Флаги кодов условий				
		X	N	Z	V	C
1100 RRR1 0000 0rrr 1100 RRR1 0000 1rrr	d ← [s] + [d] + [X] BCD-сложение	x	u	x	u	x
1101 DDD1 SSEE EEEE 1101 DDD0 SSee eeee	d ← [Dn] + [d] Dn ← [s] + [Dn]	x x	x x	x x	x x	x x
1101 AAA0 11ee eeee 1101 AAA1 11ee eeee	An ← [s] + [An]		x	x		
0000 0110 SSEE EEEE	d ← s + [d]	x	x	x	x	x
0101 QQQ0 SSEE EEEE	d ← QQQ + [d]	x	x	x	x	x
1101 RRR1 SS00 0rrr 1101 RRR1 SS00 1rrr	d ← [s] + [d] + [X] Сложение с множественной точностью	x	x	x	x	x
1100 DDD1 SSEE EEEE 1100 DDD0 SSee eeee	d ← [Dn] ∧ [d]		x	x	0	0
0000 0010 SSEE EEEE	d ← s ∧ [d]		x	x	0	0
1110 rrr1 SS10 0 DDD 1110 QQQ1 SS00 0DDD 1110 0001 11EE EEEE		x	x	x	x	x
1110 rrr0 SS10 0 DDD 1110 QQQ0 SS00 0DDD 1110 0000 11EE EEEE		x	x	x	x	x
0000 rrr1 01EE EEEE 0000 1000 01EE EEEE 0000 rrr1 01EE EEEE 0000 1000 01EE EEEE	Z ← (bit# из d); затем дополнение проверяемого бита в d			x		
0000 rrr1 10EE EEEE 0000 1000 10EE EEEE 0000 rrr1 10EE EEEE 0000 1000 10EE EEEE	Z ← (bit# из d); затем очистка проверяемого бита в d			x		

Таблица В.4 (*продолжение*)

Мнемоническое обозначение (Название)	Размер	Режим адресации	Режим адресации													
			Dn	An	(An)	(An)+	–(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR или CCR
BSET* (Проверка бита и его установка)	B L	bit# = [Dn] d = bit# = Immed d = bit# = [Dn] d = bit# = Immed d =	 x x		x x	x x	x x	x x	x x	x x	x x				
BTST* (Проверка бита)	B L	bit# = [Dn] d = bit# = Immed d = bit# = [Dn] d = bit# = Immed d =	 x x		x x	x x	x x	x x	x x	x x	x x				
CHK (Проверка регистра)	W	d = Dn s =	x		x	x	x	x	x	x	x	x	x	x	
CLR (Очистка)	B,W,L	d =	x		x	x	x	x	x	x	x				
CMP (Сравнение)	B,W,L	d = Dn s =	x	x	x	x	x	x	x	x	x	x	x	x	
CMPA (Сравнение адресов)	W L	d = An s = d = An s =	x x	x x	x x	x x	x x	x x	x x	x x	x x	x x	x x	x x	
CMPI (Сравнение с непосредственно заданным операндом)	B,W,L	s = Immed d =	x		x	x	x	x	x	x	x				
CMPM (Сравнение с операндом в памяти)	B,W,L	s = (An)+ d =				x									
DIVS (Деление со знаком)	W	d = Dn s =	x		x	x	x	x	x	x	x	x	x	x	
DIVU (Деление без знака)	W	d = Dn s =	x		x	x	x	x	x	x	x	x	x	x	
EOR (Логическое Исключающее ИЛИ)	B,W,L	s = Dn d =	x		x	x	x	x	x	x	x				
EORI (Исключающее ИЛИ с непосредственно заданным операндом)	B,W,L	s = Immed d =	x		x	x	x	x	x	x	x				x
EXG (Обмен)	L	s = Dn d = s = An d =	x x	x x											
EXT (Расширение знака)	W L	d = d =	x x												

Код операции $b_{15}...b_0$	Выполняемая операция	Флаги кодов условий				
		X	N	Z	V	C
0000 rrr1 11EE EEEE 0000 1000 11EE EEEE 0000 rrr1 11EE EEEE 0000 1000 11EE EEEE	Z ← $\overline{\text{(bit\# из d)}}$; затем установить в 1 проверяемый бит из d			x		
0000 rrr1 00EE EEEE 0000 1000 00EE EEEE 0000 rrr1 00EE EEEE 0000 1000 00EE EEEE	Z ← $\overline{\text{(bit\# из d)}}$;			x		
0100 DDD1 10ee eeee	if [Dn] < 0 or [Dn] > [s], then сгенерировать прерывание		x	u	u	u
0100 0010 SSEE EEEE	d ← 0		0	1	0	0
1011 DDD0 SSee eeee	[d] − [s]		x	x	x	x
1011 AAA0 11ee eeee 1011 AAA1 11ee eeee	[An] − [s]		x	x	x	x
0000 1100 SSEE EEEE	[d] − [s]		x	x	x	x
1011 RRR1 SS00 1rrr	[d] − [s]		x	x	x	x
1000 DDD1 11ee eeee	d ← [d] ÷ [s], используя 32 бита из d и 16 из s		x	x	x	0
1000 DDD0 11ee eeee	d ← [d] ÷ [s], используя 32 бита из d и 16 из s		x	x	x	0
1011 rrr1 SSEE EEEE	d ← [Dn] ⊕ [d]		x	x	0	0
0000 1010 SSEE EEEE	d ← s ⊕ [d]		x	x	0	0
1100 DDD1 0100 0DDD 1100 AAA1 0100 1AAA 1100 DDD1 1000 1AAA	[s] ↔ [d]					
0100 1000 1000 0DDD 0100 1000 1100 0DDD	(биты 15–8 из d) ← (бит 7 из d) (биты 31–16 из d) ← (бит 15 из d)		x x	x x	0 0	0 0

Таблица В.4 (*продолжение*)

Мнемоническое обозначение (название)	Размер	Режим адресации		Режим адресации												
				Dn	An	(An)	(An)+	–(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR или CCR
JMP (Переход)			d =			x		x	x	x	x	x	x			
JSR (Переход к подпрограмме)			d =			x		x	x	x	x	x	x			
LEA (Загрузка исполнительного адреса)	L	d = An	s =			x		x	x	x	x	x	x			
LINK (Связывание и резервирование)		disp = immed	s =		x											
LSL (Логический сдвиг влево)	B,W,L	count = [Dn]	d =	x												
		count = QQQ	d =	x												
	W	count = 1	d =			x	x	x	x	x	x	x				
LSR (Логический сдвиг вправо)	B,W,L	count = [Dn]	d =	x												
		count = QQQ	d =	x												
	W	count = 1	d =			x	x	x	x	x	x	x				
MOVE (Пересылка)	B,W,L	s = Dn	d =	x		x	x	x	x	x	x	x				
		s = An	d =	x		x	x	x	x	x	x	x				
		s = (An)	d =	x		x	x	x	x	x	x	x				
		s = (An)+	d =	x		x	x	x	x	x	x	x				
		s = –(An)	d =	x		x	x	x	x	x	x	x				
		s = d(An)	d =	x		x	x	x	x	x	x	x				
		s = d(An,Xi)	d =	x		x	x	x	x	x	x	x				
		s = Abs.W	d =	x		x	x	x	x	x	x	x				
		s = Abs.L	d =	x		x	x	x	x	x	x	x				
		s = d(PC)	d =	x		x	x	x	x	x	x	x				
		s = d(PC,Xi)	d =	x		x	x	x	x	x	x	x				
		s = immed	d =	x		x	x	x	x	x	x	x				
	W	d = CCR	s =	x		x	x	x	x	x	x	x	x	x	x	
		d = SR	s =	x		x	x	x	x	x	x	x	x	x	x	
		s = SR	d =	x		x	x	x	x	x	x	x				
	L	s = SP	d =		x											
		d = SP	s =		x											
MOVEA (Пересылка адреса)	W,L	d = An	s =	x	x	x	x	x	x	x	x	x	x	x	x	

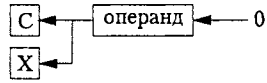
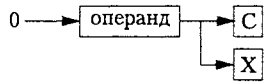
Код операции $b_{15}...b_0$	Выполняемая операция	Флаги кодов условий				
		X	N	Z	V	C
0100 1110 11EE EEEE	PC ← исполнительный адрес d					
0100 1110 10EE EEEE	SP ← [SP] − 4; [SP] ← [PC]; PC ← исполнительный адрес d					
0100 AAA111ee eeee	An ← исполнительный адрес d					
0100 1110 0101 0AAA	SP ← [SP] − 4; [SP] ← [An]; An ← [SP]; SP ← [SP] + disp					
1110 rrr1 SS10 1DDD 1110 QQQ1 SS00 1DDD 1110 0011 11EE EEEE		x	x	x	0	x
1110 rrr0 SS10 1DDD 1110 QQQ0 SS00 1DDD 1110 0010 11EE EEEE		x	x	x	0	x
00SS RRRM MMee eeee	d ← [s]		x	x	0	0
0100 0100 11ee eeee	CCR ← [биты 7–0 из s]	x	x	x	x	x
0100 0110 11ee eeee	SR ← [s]	x	x	x	x	x
0100 0000 11EE EEEE	d ← [SR]					
0100 1110 0110 1AAA	d ← [SP]					
0100 1110 0110 0AAA	SP ← [d]					
00SS AAA0 01ee eeee	An ← s					

Таблица В.4 (*продолжение*)

Мнемоническое обозначение (название)	Размер	Режим адресации	Режим адресации												
			Dn	An	(An)	(An)+	–(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR или CCR
MOVEM* (Пересылка нескольких регистров)	W	s = Xn d =			x		x	x	x	x	x				
		d = Xn s =			x	x		x	x	x	x	x	x		
	L	s = Xn d =			x		x	x	x	x	x				
		d = Xn s =			x	x		x	x	x	x	x	x		
MOVEP* (Пересылка периферийных данных)	W	s = Dn d =						x							
	L	s = Dn d =						x							
	W	s = d(An) d =	x												
	L	s = d(An) d =	x												
MOVEQ (Быстрая пересылка)	L	s = immed8 d =	x												
MULS (Умножение со знаком)	W	d = Dn s =	x		x	x	x	x	x	x	x	x	x	x	
MULU (Умножение без знака)	W	d = Dn s =	x		x	x	x	x	x	x	x	x	x	x	
NBCD (Отрицание BCD)	B	d =	x		x	x	x	x	x	x	x				
NEG (Отрицание)	B,W,L	d =	x		x	x	x	x	x	x	x				
NEGX (Отрицание расширенное)	B,W,L	d =	x		x	x	x	x	x	x	x				
NOP (Нет операции)															
NOT (Дополнение)	B,W,L	d =	x		x	x	x	x	x	x	x				
OR (Логическое ИЛИ)	B,W,L	s = Dn d =			x	x	x	x	x	x	x				
		d = Dn s =	x		x	x	x	x	x	x	x	x	x	x	
ORI (Логическое ИЛИ с непосредственно заданным операндом)	B,W,L	s = immed d =	x		x	x	x	x	x	x	x				x
PEA (Проталкивание в стек исполнительного адреса)	L	s =			x			x	x	x	x	x	x		

Код операции $b_{15}...b_0$	Выполняемая операция	Флаги кодов условий				
		X	N	Z	V	C
0100 1000 10EE EEEE 0100 1100 10ee eeee 0100 1000 11EE EEEE 0100 1100 11ee eeee	d ← [Xn] Xn ← [s] d ← [Xn] Xn ← [s] } Второе слово используется для определения регистров					
0000 DDD1 1000 1AAA 0000 DDD1 1100 1AAA 0000 DDD1 0000 1AAA 0000 DDD1 0100 1AAA	Четные (нечетные) байты из d ← [Dn] Dn ← Четные (нечетные) байты из d					
0111 DDD0 QQQQ QQQQ	Dn ← QQQQQQQQ		x	x	0	0
1100 DDD1 11ee eeee	Dn ← [s] × [Dn]		x	x	0	0
1100 DDD0 11ee eeee	Dn ← [s] × [Dn]		x	x	0	0
0100 1000 00EE EEEE	d ← 0 − [d] − [X], используя арифметику BCD	x	u	x	u	x
0100 0100 SSEE EEEE	d ← 0 − [d]	x	x	x	x	x
0100 0000 SSEE EEEE	d ← 0 − [d] − [X]	x	x	x	x	x
0100 1110 0111 0001	Нету					
0100 0110 SSEE EEEE	d ← [d]		x	x	0	0
1000 DDD1 SSEE EEEE 1000 DDD0 SSee eeee	d ← [s] ∨ [d]		x	x	0	0
0000 0000 SSEE EEEE	d ← s ∨ [d]		x	x	0	0
0100 1000 01ee eeee	SP ← [SP] − 4; [SP] ← исполнительный адрес s					

Таблица В.4 (*продолжение*)

Мнемоническое обозначение (название)	Размер	Режим адресации	Режим адресации												
			Dn	An	(An)	(An)+	−(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR или CCR
RESET (Сброс)															
ROL (Циклический сдвиг влево без учета X)	B,W,L W	count = [Dn] d = count = QQQ d = count = 1 d =	x x		x	x	x	x	x	x	x				
ROR (Циклический сдвиг вправо без учета X)	B,W,L W	count = [Dn] d = count = QQQ d = count = 1	x x		x	x	x	x	x	x	x				
ROXL (Циклический сдвиг влево с учетом X)	B,W,L W	count = [Dn] d = count = QQQ d = count = 1 d =	x x		x	x	x	x	x	x	x				
ROXR (Циклический сдвиг вправо с учетом X)	B,W,L W	count = [Dn] d = count = QQQ d = count = 1 d =	x x		x	x	x	x	x	x	x				
RTE (Возврат из исключения)															
RTR (Возврат и восстановление CCR)															
RTS (Возврат из подпрограммы)															
SBCD (Вычитание BCD)	B	s = Dn d = s = −(An) d =	x				x								
Scc (Установка по условию)	B	d =	x		x	x	x	x	x	x	x				
STOP (Загрузка SR и останов)		s =												x	

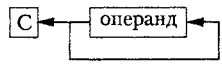
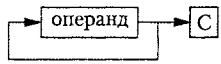
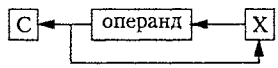
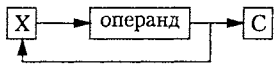
Код операции $b_{15}...b_0$	Выполняемая операция	Флаги кодов условий				
		X	N	Z	V	C
0100 1110 0111 0000	Установка выходной линии RESET					
1110 rrr1 SS11 1DDD 1110 QQQ1 SS01 1DDD 1110 0111 11EE EEEE			x	x	0	x
1100 rrr1 SS11 1DDD 1110 QQQ0 SS01 1DDD 1110 0111 11EE EEEE			x	x	0	x
1110 rrr1 SS11 0DDD 1110 QQQ1 SS01 0DDD 1110 0101 11EE EEEE		x	x	x	0	x
1100 rrr0 SS11 0DDD 1110 QQQ0 SS01 0DDD 1110 0100 11EE EEEE		x	x	x	0	x
0100 1110 0111 0011	SR ← [[SP]]; SP ← [SP] + 2; PC ← [[SP]]; SP ← [SP] + 4;	x	x	x	x	x
0100 1110 0111 0111	CCr ← [[SP]]; SP ← [SP] + 2; PC ← [[SP]]; SP ← [SP] + 4;	x	x	x	x	x
0100 1110 0111 0101	PC ← [[SP]]; SP ← [SP] + 4					
1000 RRR1 0000 0rrr 1000 RRR1 0000 1rrr	d ← [d] − [s] − [X] Вычитание BCD	x	u	x	u	x
0101 CCCC 11EE EEEE	Установка всех 8 битов d в 1, если cc истинно; в противном случае — заполнение их нулями.					
0100 1110 0111 0010	SR ← s; ожидание прерывания	x	x	x	x	x

Таблица В.4 (*продолжение*)

Мнемоническое обозначение (название)	Размер	Режим адресации	Режим адресации												
			Dn	An	(An)	(An)+	–(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR или CCR
SUB (Вычитание)	B,W,L	s = Dn d =	x		x	x	x	x	x	x	x				
		d = Dn s =	x	x	x	x	x	x	x	x	x	x	x	x	
SUBA (Вычитание адреса)	W	d = An s =	x	x	x	x	x	x	x	x	x	x	x	x	
	L	d = An s =	x	x	x	x	x	x	x	x	x	x	x	x	
SUBI (Вычитание с непосредственно заданным операндом)	B,W,L	s = immed d =	x		x	x	x	x	x	x	x				
SUBQ (Быстрое вычитание)	B,W,L	s = immed3 d =	x	x	x	x	x	x	x	x	x				
SUBX (Вычитание расширенное)	B,W,L	s = Dn d =	x												
		s = –(An) d =					x								
SWAP (Перестановка содержимого первой и второй половины регистра)	W	d =	x												
TAS (Проверка и установка)	B	d =	x		x	x	x	x	x	x	x				
TRAP (Ловушка)		s = immed4													
TRAPV (Ловушка по переполнению)															
TST (Проверка)	B,W,L	d =	x		x	x	x	x	x	x	x				
UNLK (Удаление связи)				x											

Код операции $b_{15}...b_0$	Выполняемая операция	Флаги кодов условий				
		X	N	Z	V	C
1001 DDD1 SSEE EEEE 1001 DDD0 SSee eeee	d ← [d] − [s]	x	x	x	x	x
1001 AAA0 11ee eeee 1001 AAA1 11ee eeee	An ← [An] − [s]					
0000 0100 SSEE EEEE	d ← [d] − s	x	x	x	x	x
0101 QQQ1 SSEE EEEE	d ← [d] − QQQ	x	x	x	x	x
1001 RRR1 SS00 0rrr 1001 RRR1 SS00 1rrr	d ← [d] − [s] − [X]	x	x	x	x	x
0100 1000 0100 0DDD	$[Dn]_{31-16} \leftrightarrow [Dn]_{15-0}$		x	x	0	0
0100 1010 11EE EEEE	Проверка значения d и установка флагов N и Z; установка бита 7 из d в 1		x	x	0	0
0100 1110 0100 VVVV	SP ← [SP] − 4; [SP] ← [PC]; SP ← [SP] − 2; [SP] ← [SR]; PC ← вектор					
0100 1110 0111 0110	If V = 1, then SP ← [SP] − 4; [SP] ← [PC]; SP ← [SP] − 2; [SP] ← [SR]; PC ← TRAPV вектор					
0100 1010 SSEE EEEE	Проверка значения d и установка флагов N и Z		x	x	0	0
0100 1110 0101 1AAA	SP ← [An]; An ← [[SP]]; SP ← [SP] + 4					

Таблица В.5. Команды перехода процессора 68000

Мнемоническое обозначение (название)	Величина смещения	Код операции	Выполняемая операция
BRA (Переход всегда)	8 16	0100 0000 PPPP PPPP 0110 0000 0000 0000 PPPP PPPP PPPP PPPP	PC ← [PC] + disp
Bcc (Переход по условию)	8 16	0110 CCCC PPPP PPPP 0110 CCCC 0000 0000 PPPP PPPP PPPP PPPP	If cc истина, then PC ← [PC] + disp
BSR (Переход к подпрограмме)	8 16	0110 0001 PPPP PPPP 0110 0001 0000 0000 PPPP PPPP PPPP PPPP	SP ← [SP] −4; [SP] ← [PC]; PC ← [PC] + disp

DBcc (Декремент и условный переход)	16	0101 CCCC 1100 1DDD
		PPPP PPPP PPPP PPPP
			If cc ложь, then
			Dn ← [Dn] − 1;
			If [Dn] ≠ −1, then
			PC ← [PC] + disp

DBRA (Декремент и переход)	Ассемблер интерпретирует эту команду как DBF (см. строку DBcc)		

Таблица В.6. Коды условий для команд Bcc, DBcc и Scc

Машинный код CCCC	Суффикс условия cc	Название	Проверяемое условие
0000	T	Истина	Всегда истина
0001	F	Ложь	Всегда ложь
0010	HI	Старше	$C \vee Z = 0$
0011	LS	Младше или равно	$C \vee Z = 1$
0100	CC	Очистка переноса	$C = 0$
0101	CS	Установка переноса	$C = 1$
0110	NE	Не равно	$Z = 0$
0111	EQ	Равно	$Z = 1$
1000	VC	Очистка переполнения	$V = 0$
1001	VS	Установка переполнения	$V = 1$
1010	PL	Плюс	$N = 0$
1011	MI	Минус	$N = 1$
1100	GE	Больше или равно	$N \oplus V = 0$
1101	LT	Меньше	$N \oplus V = 1$
1110	GT	Больше	$Z \vee (N \oplus V) = 0$
1111	LE	Меньше или равно	$Z \vee (N \oplus V) = 1$

Приложение Г

Система команд процессоров IA-32

О системе команд Intel IA-32 рассказывалось, как вы помните, в главе 3. Этот набор весьма обширен. Наш обзор охватывает только небольшую его часть — около 50 команд, включая те, которые рассмотрены в главе 3. Однако в приложении содержатся и характеристики других типов команд.

Структура регистров IA-32 показана на рис. 3.37 и 3.38, а описана в разделе 3.16.1. На рис. 3.41 представлен общий формат команд IA-32. В этой архитектуре память адресуется побайтово, а адреса имеют длину 32 бита. Поддерживаются два размера операндов: двойное слово (32 бита) и байт (8 бит). В ранних версиях процессоров Intel использовались операнды длиной в одно слово (16 бит). Процессоры IA-32 могут функционировать и в 16-разрядном режиме.

Г.1. Кодирование команд

На рис. Г.1, *а* показан общий формат команд IA-32. Код операции занимает один или два байта. Для некоторых команд он расширяется до 3-битового поля Reg/OPcode байта ModR/M (рис. Г.1, *б*). Поскольку коды операций не имеют единого формата, мы не будем обсуждать детали их структуры. В разделе Г.3 описываются префиксные байты, которые в форматах большинства рассматриваемых здесь команд не используются.

Команда может занимать от 1 (только код операции) до 11 и более байт, если задаются 4-байтовое смещение и 4-байтовый операнд в режиме непосредственной адресации, а также 2 байта адресного режима и код операции. Например, для команд инкремента (INC) и декремента (DEC) с операндом в регистре достаточно 1-байтового кода операции, включающего 3-битовое поле для имени регистра. А вот пример длинной команды, занимающей 11 байт:

```
MOV  DWORD PTR [EBP + ESI*4 + DISP],10
```

Структура приведенной команды рассматривалась в разделе 3.17.1. Там обсуждались и другие примеры команд.

Размер данных операнда (8 или 32 бита) указывается в коде операции. Кроме того, в нем фиксируется, задан ли один из операндов непосредственно в команде (в этом случае его значение содержится в последнем поле команды).

Как минимум один операнд из двух, которые используются в команде, должен находиться в регистре. Регистр задается в поле Reg/OPcode байта ModR/M.

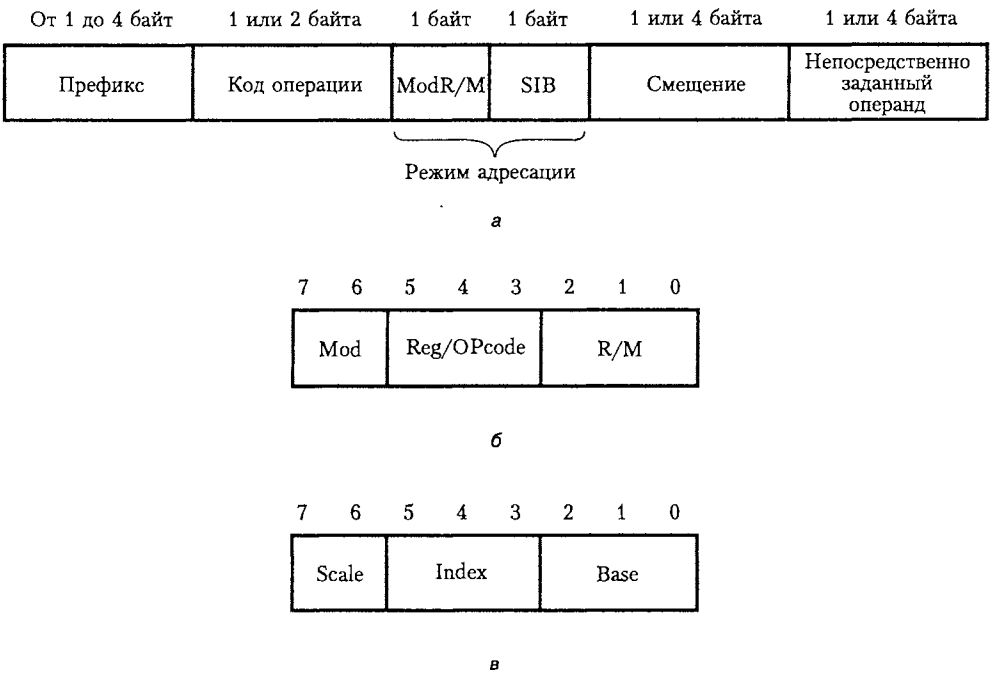


Рис. Г.1. Формат команд IA-32: общий формат (*а*); байт ModR/M (*б*); байт SIB (*в*)

В табл. Г.1 вам предоставляется возможность ознакомиться с 3-битовыми кодами регистров. Если второй операнд тоже содержится в регистре, этот регистр задается в поле R/M того же байта. Если же второй операнд находится за пределами регистра, он может быть непосредственно задан в команде или же храниться в памяти. Адрес операнда в памяти указывается при помощи двух байтов режима адресации и поля смещения, о чем вы узнаете из следующего раздела. Как рассказывалось в разделе 3.17.1, в поле кода операции имеется бит *направления*, определяющий, какой из двух операндов команды является исходным.

Таблица Г.1. Кодировка поля регистра в командах IA-32

Поле Reg/Base/Index*	Регистр
000	EAX
001	ECX
010	EDX
011	EBX
100	ESP
101	EBP
110	ESI
111	EDI

* Регистр ESP (100) нельзя использовать в качестве индексного

Г.1.1. Режимы адресации

В табл. 3.3 перечислены режимы адресации, которые поддерживаются архитектурой IA-32, приведен синтаксис языка ассемблера, используемый для их обозначения, а также указан способ формирования исполнительного адреса. Мы уже рассмотрели раньше режим непосредственной адресации и способы применения поля Reg/OPcode байта ModR/M для задания регистра, в котором расположен один из операндов. В табл. Г.2 показано, как задается второй операнд. Такие типы адресации, как косвенная регистровая, базовая со смещением, регистровая, определяются 2-битовым полем Mod в байте ModR/M, о чем свидетельствуют первые четыре строки таблицы. В 3-битовом поле R/M обычно задается регистр (Reg), используемый в этих режимах. Исключения из правила существуют в режимах адресации, перечисленных во второй части таблицы. Во всех режимах, кроме режима прямой адресации, применяется байт SIB (рис. Г.1, *в*). В этом байте задаются базовый и индексный регистры. Коды коэффициентов масштабирования 1, 2, 4 и 8 приведены в табл. Г.3.

Таблица Г.2. Режимы адресации IA-32, выбираемые байтами ModR/M и SIB

Байты ModR/M		**Адресация**
Поле Mod $b_7\ b_6$	**Поле R/M** $b_2\ b_1\ b_0$	
0 0	Reg	Косвенная регистровая, EA = [Reg]
0 1	Reg	Базовая с 8-разрядным смещением, EA = [Reg] + Disp8
1 0	Reg	Базовая с 32-разрядным смещением, EA = [Reg] + Disp32
1 1	Reg	Регистровая, EA = Reg
Исключения		
0 0	1 0 1	Прямая, EA = Disp32
0 0	1 0 0	Базовая индексная (применяется байт SIB), EA = [Base] + [Index] × Scale. Если Base = EBP, то устанавливается режим индексной адресации с 32-разрядным смещением, EA = [Index] × Scale + Disp32
0 1	1 0 0	Базовая индексная с 8-разрядным смещением (применяется байт SIB), EA = [Base] + [Index] × Scale + Disp8
1 0	1 0 0	Базовая индексная с 32-разрядным смещением (применяется байт SIB), EA = [Base] + [Index] × Scale + Disp32

Таблица Г.3. Кодирование поля коэффициента масштабирования в байте SIB IA-32

Поле	Коэффициент масштабирования
0 0	1
0 1	2
1 0	4
1 1	8

Как указано в табл. Г.3, регистр ESP (код 100) не может использоваться в качестве индексного. Причина ограничения в том, что данный регистр содержит указатель на стек процессора. Когда в поле индекса байта SIB помещается битовый код 100, масштабируемый индекс в команде не учитывается, но остальные компоненты режима адресации функционируют обычным образом. Представленная схема имеет один полезный эффект. Из табл. Г.2 видно, что в первых трех режимах адресации не может использоваться регистр ESP, поскольку код в этом регистре обозначает исключения, соответствующие последним трем режимам в таблице. Однако если значение 100 помещается и в поле базы и в поле индекса байта SIB, то устанавливаются первые три режима адресации из указанных в табл. Г.2, когда регистр ESP используется в качестве базового, поскольку индекс в описанном случае не требуется.

Обратите внимание на то, что содержимое базового регистра 101 (EBP) выступает как исключение при базовой индексной адресации: с его помощью осуществляется переход в режим индексной адресации с 32-разрядным смещением. Регистр EBP может использоваться в этом режиме в качестве базового, для чего в режиме базовой индексной адресации со смещением должно быть задано нулевое смещение.

Г.2. Основные команды

В табл. Г.4 приведен упорядоченный по алфавиту список наиболее часто используемых команд IA-32. В эту таблицу включены все команды, описанные в главе 3, за исключением команд перехода и специализированных строковых команд с опцией повторения, которые обсуждались в разделе 3.21.3. Обзор команд условного и безусловного перехода дан ниже, в двух подразделах. Информация о строковых командах приведена в разделе Г.4. Первый столбец табл. Г.4 содержит мнемонические обозначения и названия команд. Во втором столбце указаны размеры операндов, которые могут использоваться в каждой из команд: B (байт) или D (32-разрядное двойное слово). В третьем столбце перечислены возможные местонахождения исходного и результирующего операндов, обозначенные как:

reg — один из восьми регистров процессора;

mem — память;

imm — 8- или 32-разрядный операнд, заданный непосредственно;

imm8 — 8-разрядный операнд, заданный непосредственно.

Четвертый столбец включает данные о выполняемой командой операции. В последнем столбце показано, как команда изменяет значения флагов кодов условий:

x — воздействует;

0 — устанавливает в 0;

1 — устанавливает в 1;

«пусто» — не воздействует;

? — непредсказуемо.

Таблица Г.4. Команды IA-32

Мнемоническое обозначение (название)	Размер	Операнды		Выполняемая операция	Флаги кодов условий			
		dst	src		S	Z	O	C
ADC (Сложение с переносом)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] + [src] + [CF]	x	x	x	x
ADD (Сложение)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] + [src]	x	x	x	x
AND (Логическое И)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] ∧ [src]	x	x	0	0
BT (Проверка битов)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# из [dst]				x
BTC (Проверка битов и дополнение)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# из [dst]; дополнение bit# из [dst]				x
BTR (Проверка битов и переустановка)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# из [dst]; очистка bit# из [dst] нулями				x
BTS (Проверка битов и установка)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# из [dst]; установка bit# из [dst] в 1				x

Таблица Г.4 (*продолжение*)

Мнемоническое обозначение (название)	Размер	Операнды		Выполняемая операция	Флаги кодов условий			
		dst	src		S	Z	O	C
CALL (Вызов подпрограммы)	D	reg mem		ESP ← [ESP] − 4; [ESP] ← [EIP]; EIP ← исполнительный адрес dst				
CLC (Очистка переноса)				CF ← 0				0
CLI (Очистка целочисленного флага)				IF ← 0				
CMC (Дополнение и перенос)				CF ← [CF]				x
CMP (Сравнение)	B,D	reg reg mem reg mem	reg mem reg imm imm	[dst] − [src]	x	x	x	x
DEC (Декремент)	B,D	reg mem		dst ← [dst] − 1	x	x	x	
DIV (Беззнаковое деление)	B,D		reg mem	Для B: [AL]/[src]; AL ← частное; AH ← остаток Для D: [EAX]/[src]; EAX ← частное; EDX ← остаток	?	?	?	?
HLT (Останов)				Приостанавливает выполнение до сброса или внешнего прерывания				
IDIV (Деление со знаком)	B,D		reg mem	Для B: [AL]/[src]; AL ← частное; AH ← остаток Для D: [EAX]/[src]; EAX ← частное; EDX ← остаток	?	?	?	?

Мнемоническое обозначение (название)	Размер	Операнды dst	Операнды src	Выполняемая операция	Флаги кодов условий S	Z	O	C
IMUL (Умножение со знаком)	B,D		reg mem	(произведение двойной длины) для B: AX ← [AL] × [src]; для D: EDX,EAX ← [EAX] × [src]	?	?	x	x
	D	reg reg	reg mem	(произведение одинарной длины) reg ← [reg] × [src]	?	?	x	x
IN (Изолированный ввод)	B,D	dst = AL или EAX src = imm8 или [DX]		AL или EAX ← [src]				
INC (Инкремент)	B,D	reg mem		dst ← [dst] + 1	x	x	x	
INT (Программное прерывание)	D		imm8	Push EFLAGS; Push EIP; EIP ← адрес (определяется по imm8)				
IRET (Возврат из прерывания)	D			Pop EIP; Pop EFLAGS	x	x	x	x
LEA (Загрузка исполнительного адреса)	D	reg	mem	reg ← исполнительный адрес src				
LOOP (Цикл)	D	target		ECX ← [ECX] − 1; If ([ECX] ≠ 0) EIP ← target				
LOOPE (Цикл по равенству/нулю)	D	target		ECX ← [ECX] − 1; If ([ECX] ≠ 0 ∧ [Z] = 1) EIP ← target				
LOOPNE (Цикл по неравенству/не нулю)	D	target		ECX ← [ECX] − 1; If ([ECX] ≠ 0 ∧ [Z] ≠1) EIP ← target				
MOV (Пересылка)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [src]				

Таблица Г.4 (*продолжение*)

Мнемоническое обозначение (название)	Размер	Операнды		Выполняемая операция	Флаги кодов условий			
		dst	src		S	Z	O	C
MOVSX (Расширение знака байта до количества разрядов регистра)	B	reg reg	reg mem	reg ← [src] с расширенным знаком				
MOVZX (Расширение байта нулями до количества разрядов регистра)	B	reg reg	reg mem	reg ← [src] расширенное нулями				
MUL (Беззнаковое умножение)	B,D		reg mem	(Произведение двойной длины) Для B: AX ← [AL] × [src]; Для D: EDX,EAX ← [EAX] × [src]	?	?	x	x
NEG (Отрицание)	B,D	reg mem		dst ← дополнение [dst] до двух	x	x	x	x
NOP (Нет операции)				Псевдоним для XCHG EAX,EAX				
NOT (Логическое дополнение)	B,D	reg mem		dst ← $\overline{[dst]}$				
OR (Логическое ИЛИ)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] ∨ [src]	x	x	0	0
UOT (Изолированный вывод)	B,D	dst = imm8 или [DX] src = AL или EAX		dst ← [AL] или [EAX]				
POP (Выталкивание из стека)	D	reg mem		dst ← [[ESP]]; ESP ← [ESP] + 4				
POPAD (Выталкивание из стека во все регистры, кроме ESP)	D			Выталкивание 8 двойных слов из стека в EDI, ESI, EBP, игнорирование, EBX, EDX, ECX, EAX; ESP ← [ESP]+ 32				

Мнемоническое обозначение (название)	Размер	Операнды		Выполняемая операция	Флаги кодов условий			
		dst	src		S	Z	O	C
PUSH (Проталкивание в стек)	D		reg mem imm	ESP ← [ESP] − 4; [ESP] ← [src]				
PUSHAD (Проталкивание в стек всех регистров)	D			Проталкивание в стек содержимого регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; ESP ← [ESP] − 32				
RCL (Циклический сдвиг влево с флагом C)	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.32, *б*; в операнде src задается количество разрядов сдвига			?	x
RCR (Циклический сдвиг вправо с флагом C)	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.32, *г*; в операнде src задается количество разрядов сдвига			?	x
RET (Возврат из подпрограммы)				EIP ← [[ESP]]; ESP ← [ESP] + 4				
ROL (Циклический сдвиг влево)	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.32, *а*; в операнде src задается количество разрядов сдвига			?	x
ROR (Циклический сдвиг вправо)	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.32, *в*; в операнде src задается количество разрядов сдвига			?	x
SAL (Арифметический сдвиг влево), то же что SHL	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.30, *а*; в операнде src задается количество разрядов сдвига	x	x	?	x
SAR (Арифметический сдвиг вправо)	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.30, *в*; в операнде src задается количество разрядов сдвига	x	x	?	x
SBB (Вычитание с заимствованием)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] − [src] − [CF]	x	x	x	x
SHL (Сдвиг влево), то же что SAL	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.30, *а*; в операнде src задается количество разрядов сдвига	x	x	?	x

Таблица Г.4 (*продолжение*)

Мнемоническое обозначение (название)	Размер	Операнды		Выполняемая операция	Флаги кодов условий			
		dst	src		S	Z	O	C
SHR (Сдвиг вправо)	B,D	reg reg mem mem	imm8 CL imm8 CL	См. рис. 2.30, *б*; в операнде src задается количество разрядов сдвига	x	x	?	x
STC (Установка флага переноса)				CF ← 1				1
STI (Установка флага прерывания)				IF ← 1				
SUB (Вычитание)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] − [src]	x	x	x	x
TEST (Проверка)	B,D	reg mem reg mem	reg reg imm imm	[dst] ∧ [src]; установка флагов в зависимости от результата	x	x	0	0
XCHG (Обмен)	B,D	reg reg	reg mem	[reg] ↔ [src]				
XOR (Исключающее ИЛИ)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] ⊕ [src]	x	x	0	0

Г.2.1. Команды условного перехода

Обзор команд условного перехода дан в табл. Г.5. В программах на языке ассемблера целевой адрес может задаваться непосредственно (см. раздел 3.19.1). Но машинная команда содержит число со знаком (смещение), определяющее расстояние до целевого адреса в байтах относительно обновленного содержимого счетчика команд. Используются два размера смещения: 1 байт и 4 байта. Ассемблер вычисляет смещение во время трансляции исходной программы в машинный код.

Г.2.2. Команды безусловного перехода

В разделе 3.19.1 рассматривалась команда безусловного перехода JMP. Для определения целевого адреса в ней могут применяться любые режимы адресации. За счет этого реализуется разветвление программы в нескольких направлениях, что напоминает действие команды CASE в языках высокого уровня.

Таблица Г.5. Команды условного перехода IA-32

Мнемоническое обозначение	Название условия	Проверка кода условия
JS	Знак (отрицательный)	SF = 1
JNS	Знак отсутствует (положительный или нуль)	SF = 0
JE/JZ	Равно/нуль	ZF = 1
JNE/JNZ	Не равно/не нуль	ZF = 0
JO	Переполнение	OF = 1
JNO	Нет переполнения	OF = 0
JC/JB	Перенос/беззнаковое младше	CF = 1
JNC/JAE	Нет переноса/беззнаковое старше или равно	CF = 0
JA	Беззнаковое старше	CF ∨ ZF = 0
JBE	Беззнаковое младше или равно	CF ∨ ZF = 1
JGE	Со знаком больше или равно	SF ⊕ OF = 0
JL	Со знаком меньше	SF ⊕ OF = 1
JG	Со знаком больше	ZF ∨ (SF ⊕ OF) = 0
JLE	Со знаком меньше или равно	ZF ∨ (SF ⊕ OF) = 1

Г.3. Префиксные байты

Байты префикса команды (рис. Г.1, *а*) делятся на четыре группы. Хотя команда может использовать несколько префиксных байтов, из каждой группы применяется только одно значение. Первая группа включает коды *повторения*, указывающие, что заданная в команде операция должна быть повторена несколько раз. Команды, поддерживающие эту опцию, называются *строковыми* (см. раздел Г.4). В разделе 3.21.3 приведен пример повторения строковой операции с целью пересылки блока двойных слов из памяти к устройству ввода-вывода и в обратном порядке. Кроме того, префиксные байты из этой группы определяют команды потокового расширения SIMD (SSE), описанные в разделах 3.23.3, 11.3.6 и 11.3.7.

Следующие две группы включают только по одному коду, которые изменяют применяемый по умолчанию размер операнда или адрес (см. раздел Г.5).

Четвертая группа префиксных байтов служит для выбора сегментных регистров, используемых при формировании исполнительных адресов памяти. О сегментных регистрах рассказывалось в разделе 11.3.1.

Г.4. Дополнительные команды

Система команд IA-32 включает гораздо больше команд, чем указано в табл. Г.4. Здесь кратко рассказывается о четырех типах команд, не описанных в таблице.

Г.4.1. Команды обработки строк

Существует несколько команд, предназначенных для эффективного выполнения повторяющихся операций над элементами данных, расположенных в памяти по последовательным адресам. Такие структуры данных называются *строками*, а соответствующие команды — строковыми командами. Отдельные элементы строки могут быть байтами или 32-разрядными двойными словами. Самым распространенным способом применения строковых команд является перемещение строки из одного места памяти в другое или сравнение строк на предмет эквивалентности.

Работа строковых команд будет проиллюстрирована на примере команд перемещения строк. Байты и двойные слова перемещаются посредством кодов операций MOVSB и MOVSD. Они отличаются от обычных команд пересылки тем, что состоят только из кодов операций и не содержат явно заданных операндов. Предполагается, что адрес исходного операнда находится в регистре ESI, а адрес операнда назначения — в регистре EDI. В ходе выполнения команды MOVSB сначала пересылается байт из исходной позиции в место назначения, а затем увеличиваются значения регистров-указателей ESI и EDI. Заданную команду можно поместить внутрь цикла, по очереди перемещающего все байты строки. Альтернативный вариант — использование префикса повторения, позволяющего переместить целую строку одной командой. В этом случае наряду с регистрами ESI и EDI инициализируется регистр ECX, в который помещается значение длины строки. После пересылки каждого байта значение регистра уменьшается. Например, команда

```
REP  MOVSB
```

перемещает целую строку байтов. Она выбирается из памяти один раз, заданная же в ней операция выполняется многократно, пока содержимое регистра ECX не достигнет нуля.

Строковые команды с опцией повторения включены в систему команд IA-32 из соображений производительности. То же действие, которое выполняет приведенная выше команда, можно запрограммировать при помощи команды

```
MOV  BYTE PTR [EDI],[ESI]
```

помещенной внутрь цикла, где явно увеличиваются значения регистров-указателей и уменьшается значение регистра ECX, пока оно не станет равным нулю. Очевидно, что продолжительность выполнения такого цикла больше.

Г.4.2. Команды с плавающей запятой, MMX и SSE

Система команд IA-32 предполагает выполнение полного диапазона операций над данными с плавающей запятой в формате IEEE. Для хранения таких данных используются восемь регистров с плавающей запятой (рис. 3.37). Помимо операций сложения, вычитания, умножения и деления поддерживаются также тригонометрические функции.

Команды MMX, описанные в разделе 3.23.2, используются для параллельного выполнения простых арифметических и логических операций над короткими целыми числами, упакованными в 64-разрядные четверные слова, которые располо-

жены в регистрах с плавающей запятой или в памяти. Такие операции необходимы для графических приложений, а также приложений цифровой обработки сигналов.

Команды SSE, впервые появившиеся в процессоре Pentium III и дополненные в процессоре Pentium 4 (см. разделы 11.3.6 и 11.3.7), выполняют параллельные арифметические операции над числами с плавающей запятой, упакованными во множество 128-разрядных регистров процессора. Эти регистры расположены отдельно от регистров общего назначения и регистров с плавающей запятой. Отдельные элементы данных могут быть 32- или 64-битовыми числами с плавающей запятой. Команды SSE в первую очередь применяются для векторных и матричных вычислений, производимых научными приложениями. В расширениях Pentium 4 операнды могут также быть 64-разрядными целыми числами. Данные этого типа используются при шифровании и дешифрировании в защищенных приложениях.

Г.5. Обработка 16-разрядных адресов и данных

В разделах 3.16.1 и 11.3.2 отмечалось, что процессор IA-32 способен выполнять программы в режиме, где он оперирует 16-разрядными адресами и данными, использовавшимися в ранних процессорах Intel, а также в описанном в данной книге режиме, где операции могут применяться к 32-разрядным адресам и данным. В любом из указанных режимов возможна обработка байтовых операндов. При работе в 32-разрядном режиме специальный бит кода операции определяет, является операнд байтом или двойным словом; в 16-разрядном режиме тот же бит устанавливает, служит операнд байтом или 16-разрядным словом. Режим по умолчанию определяется битом в дескрипторе сегмента (см. раздел 11.3.2).

В настоящей книге во время обсуждения процессора IA-32 предполагается, что он работает в установленном по умолчанию 32-разрядном режиме. Однако этот режим можно менять для каждой команды в отдельности. С этой целью к команде должен быть добавлен префиксный байт, как показано на рис. Г.1. Префиксные байты позволяют менять используемый по умолчанию размер операнда, размер адреса либо то и другое.

Г.6. Программирование

Чтобы научиться программировать на языке ассемблера, проще обратиться к встроенному ассемблеру языка высокого уровня. С примером встроенного ассемблера вы познакомились в главе 9, где в программу на языке C были включены подпрограммы ввода-вывода на языке ассемблера. В данном разделе описываются приемы использования этой функции языка C/C++. Корпорация Microsoft разработала для данного языка компилятор, функционирующий под управлением операционной системы Windows в персональных компьютерах на основе процессоров IA-32.

На рис. Г.2 показано, как включить в программу C/C++ цикл сложения (рис. 3.40, *а*). Код ассемблерной команды помещается внутрь конструкции:

```
_asm {...}
```

Объявление данных и операции инициализации производятся на языке C/C++ в начале программы, а результат выполнения ассемблерной программы, расположенный в памяти по адресу SUM, выводится командой printf в конце программы. Мы не описываем здесь процедуры создания файла и исходной программы, ввода ее текста, компиляции и запуска, поскольку они зависят от конкретного программного окружения.

Компилятор может сгенерировать шестнадцатеричный код, подобный приведенному на рис. Г.2. Он является типичным примером двоичного кодирования команд IA-32 в формате, показанном на рис. Г.1. На рис. Г.3 представлен листинг цикла из четырех команд. Слева от ассемблерных команд на рис. Г.3, *а* вы видите шестнадцатеричное представление байтов каждой команды. На рис. Г.3, *б* и Г.3, *в* изображены детали двоичного кодирования для команд ADD и JG.

```
# include <stdio.h>

void main(void)

{
  long NUM1[5];
  long SUM;
  long N;

  NUM1[0] = 17;
  NUM1[1] = 3;
  NUM1[2] = -51;
  NUM1[3] = 242;
  NUM1[4] = 113;
  N = 5;

  _asm {
              LEA   EBX,NUM1
              MOV   ECX,N
              MOV   EAX,0
              MOV   EDI,0
STARTADD:     ADD   EAX,[EBX+EDI*4]
              INC   EDI
              DEC   ECX
              JC    STARTADD
              MOV   SUM,EAX
  }
  printf("The sum of the list values is %ld\n",SUM);
}
```

Рис. Г.2. Программа на языке ассемблера IA-32 (см. рис. 3.40), помещенная внутрь программы на языке C/C++

Машинные команды (шестнадцатеричные коды)	Команды ассемблера
03 04 BB	STARTADD: ADD EAX,[EBX = EDI*4]
47	INC EDI
49	DEC ECX
7F F9	JG STARTADD

а

Код операции	Байт ModR/M	Байт SIB
03	04	BB
00000011	00 000 100	10 111 011
ADD (двойное слово)	См. табл. Г.2	См. рис. Г.1, *в*

б

Код операции	Смещение
7F	F9
01111111	111111001
JG	
(короткое смещение)	−7

в

Рис. Г.3. Кодирование тела и команд цикла, приведенного на рис. Г.2: код тела цикла (*а*); код команды ADD (*б*); код команда JG (*в*)

Сначала рассмотрим команду ADD. Единицы в двух битах справа от кода операции означают следующее: последняя единица сообщает, что размер операнда составляет 32 бита. Предпоследняя единица свидетельствует о том, что исходный операнд содержится в памяти. Воспользовавшись табл. Г.2, можно определить, что поля Mod (00) и R/M (100) задают для расположенного в памяти операнда режим базовой индексной адресации, а поле Reg/OPcode (000) указывает, что местом назначения является регистр EAX. В поле базы (011) байта SIB определено, что базовым регистром является EBX, а в поле индекса (111) — что индексным регистром служит EDI. В поле коэффициента масштабирования (10) задан коэффициент 4.

Рассмотрим, как интерпретируется код команды JG на рис. Г.3, *в*. Первые четыре бита кода операции (0111) определяют условный переход с однобайтовым смещением, а последние четыре бита — переход по условию «больше». В байте смещения содержится число −7 в формате дополнения до двух. Это расстояние в байтах от команды, следующей за командой JG, до адреса команды ADD в начале цикла.

Использование стека процессора

В качестве указателей на стек и стековый фрейм компилятор использует регистры ESP и EBP. Кроме того, он выделяет в стеке память для переменных NUM1, SUM и N, которые объявлены в главной программе как локальные. Когда эти переменные применяются в ассемблерной части программы, как в первых двух командах программы на рис. Г.2, компилятор генерирует для доступа к ним режим базовой адресации. В роли базового регистра выступает указатель на фрейм EBP, а величина смещения выражается отрицательным значением, которое отсчитывается от вершины стека и увеличивается в направлении уменьшения адресов. Если объявить эти переменные вне главной процедуры, они будут определены как глобальные, а для доступа к ним поребуется прямая адресация.

Приложение Д

Коды символов и преобразование чисел

Д.1. Коды символов

Вся информация, которая предназначена для обработки и хранения в компьютере, должна быть закодирована при помощи двоичных кодов. Положительные и отрицательные числа представляются в одной из разновидностей двоичной системы. Самые распространенные форматы чисел, в том числе целых и чисел с плавающей запятой, описаны в главе 6.

В компьютерах, используемых главным образом для обработки деловой информации, удобно представлять числа в десятичном формате. Популярная система кодирования отдельных цифр десятичного числа, называемая двоично-десятичным кодированием (Binary Coded Decimal, BCD), приведена в табл. Д.1. Двоичные коды десятичных цифр представляют собой не что иное как первые 10 значений в 4-разрядной двоичной системе счисления. Последовательности 4-разрядных кодов с соответствующим значением бита знака могут использоваться для представления положительных и отрицательных целых чисел любого диапазона.

Таблица Д.1. Двоично-кодированные десятичные цифры

Десятичная цифра	Код BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Буквы алфавита (A–Z), операторы, знаки препинания, управляющие символы («+», «–», «/», «,», «:», «;», «LF», «CR», «EOT») и числа имеют определенные коды, предназначенные для хранения и редактирования текста, выполнения операций ввода, обработки и вывода данных в программах на языках высокого уровня. Существуют две стандартные системы кодирования — ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена

информацией) и EBCDIC (Extended Binary-Coded Decimal Interchange Code — расширенный двоично-десятичный код для обмена информацией). В ASCII каждому символу соответствует 7-битовый код, а в EBCDIC — код 8-битовый. Коды ASCII- и EBCDIC-символов приведены соответственно в табл. Д.2 и Д.3. Причем кодировка ASCII используется гораздо чаще, чем EBCDIC.

Таблица Д.2. Коды символов в системе ASCII

Позиции битов	Позиции битов 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	INQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N		n	~
1111	SI	US	/	?	O	_	o	DEL

NUL — пустой символ
SOH — начало заголовка
STX — начало текста
ETX — конец заголовка
EOT — конец текста
INQ — конец передачи
ACK — подтверждение
BEL — звуковой сигнал
BS — возврат на одну позицию
HT — горизонтальная табуляция
LF — переход к новой строке
VT — вертикальная табуляция
FF — переход к новой странице
CR — возврат каретки
SO — нижний регистр
SI — верхний регистр
DLE — завершение сеанса связи
DC1–DC4 — управление устройствами 1–4
NAK — ошибка передачи
SYN — холостые данные синхронной передачи
ETB — конец передаваемого блока
CAN — отмена
EM — конец носителя данных
SUB — подстановка
ESC — прекращение
FS — разделитель файлов
GS — разделитель групп
RS — разделитель записей
US — разделитель элементов
DEL — удаление

Формат кода: | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Таблица Д.3. Коды символов в системе EBCDIC

Позиции битов	Позиции битов 7654															
3210	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NULL				SP	&	–									0
0001							/		a	j			A	J		1
0010									b	k	s		B	K	S	2
0011									c	l	t		C	L	T	3
0100	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	LC	BS	EOB	UC					f	o	w		F	O	W	6
0111	DEL	IL	PRE	EOT					g	p	x		G	P	X	7
1000									h	q	y		H	Q	Y	8
1001									i	r	z		I	R	Z	9
1010			SM		¢	!		:								
1011					.	$	,	#								
1100					<	*	%	@								
1101					(	)	–	'								
1110					+	;	>	=								
1111					|	¬	?	"								

NULL – пусто или неопределенное значение
PF – выключение перфоратора
HT – горизонтальная табуляция
LC – строчная буква
DEL – удаление
RES – восстановление

NL – новая строка
BS – возврат на одну позицию
IL – пустой
BYP – пропустить
LF – перевод строки
EOB – конец блока

PRE – префикс
SM – установка режима
PN – включение перфоратора
RS – останов устройства чтения
UC – прописная буква
EOT – конец передачи
SP – пробел

Формат кода: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Во многих приложениях удобнее использовать 8-битовые коды, поэтому базовый код ASCII часто расширяется до 8 бит. Одним из распространенных способов расширения ASCII-кода является установка старшего бита в 0. Другой популярный способ заключается в использовании старшего бита в качестве бита четности закодированного символа.

Несколько небезынтересных замечаний относительно структуры кодов ASCII и EBCDIC. Обратите внимание, что в обеих кодировках 4 младших бита кодов десятичных цифр (0–9) являются BCD-кодами, указанными в табл. Д.1. Это позволяет выполнять над десятичными цифрами две операции. Во-первых, их можно сравнивать, чтобы определить, какая из них больше. Такое сравнение может производиться теми же логическими схемами, которые используются для выполнения стандартных арифметических операций с двоичными числами. Указанная возможность полезна для сортировки строк десятичных чисел. Во-вторых, когда последовательные 7- или 8-битовые коды введенной строки представляют десятичные числа, которые должны храниться и обрабатываться единым блоком, иногда бывает полезно удалить 3 или 4 крайних слева бита каждой цифры, сжав таким образом представление чисел в строку 4-битовых кодов BCD. Для такого сжатия (или упаковки) данных необходимы разделители, отмечающие начало и конец числа, но во многих случаях, когда для хранения данных важно использовать минимум памяти, оно обычно оправдано. Что касается символов алфавита, то их двоичные коды подобраны таким образом, чтобы удобно было сортировать текст по алфавиту.

Д.2. Десятично-двоичное преобразование

Ниже рассказывается, как преобразовать десятичное число с фиксированной запятой в его двоичный эквивалент. Десятичное значение V, представленное двоичным числом

$$B = b_n b_{n-1} \ldots b_0, b_{-1} b_{-2} \ldots b_{-m}$$

таково:

$$V(B) = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \ldots + b_0 \times 2^0 + \\ + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \ldots + b_{-m} \times 2^{-m}$$

В процессе преобразования десятичного числа с фиксированной запятой в двоичное его целая и дробная части обрабатываются отдельно. Делается это следующим образом. Сначала целая часть делится на 2. Остаток записывается в младший разряд целой части двоичного представления. Частное снова делится на 2, и остаток записывается в следующий разряд целой части двоичного представления. Этот процесс повторяется до тех пор, пока частное не станет равным 0.

Затем преобразуется дробная часть. Сначала она умножается на 2. Часть произведения слева от десятичной запятой, равная 0 или 1, записывается в старший разряд двоичного представления (крайний справа, стоящий сразу за двоичной запятой). Дробная часть произведения снова умножается на 2, в результате чего получается следующий бит двоичного представления. Процесс повторяется до достижения необходимой точности.

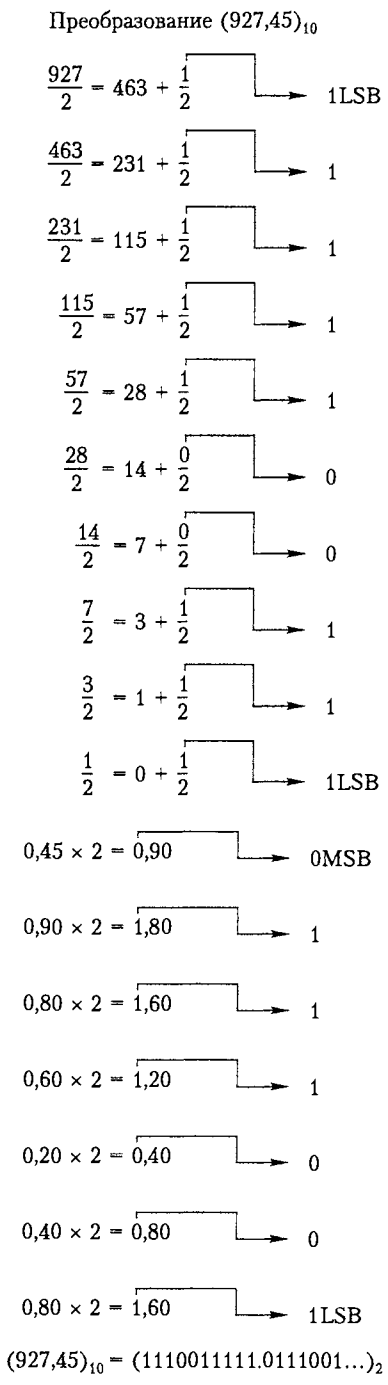


Рис. Д.1. Пример преобразования числа из десятичной системы счисления в двоичную

На рис. Д.1 приведен пример преобразования в двоичный формат десятичного числа 927,45. Обратите внимание, что преобразование целой части всегда точное, тогда как двоичная дробная часть может быть неточной. Например, не имеет точного двоичного эквивалента число $(0,45)_{10}$ — дробная часть числа, представленного на рис. Д.1. В подобных случаях двоичная дробь генерируется до заданной точности. В общем случае максимальная абсолютная погрешность e при формировании k-разрядного дробного представления лежит в пределах $e \leq 2^{-k}$. Конечно, многие десятичные дроби имеют точные двоичные представления. Например, $(0,25)_{10}$ равно $(0,01)_2$.

Алфавитный указатель

T

U

V

W

Z

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я

К. Хамахер, З. Вранешич, С. Заки

Организация ЭВМ

5-е изд.

Перевод с английского О. Здир

Редакторы	*И. Карпышенко, Е. Курбатова*
Художник	*Н. Биржаков*
Корректор	*И. Карпышенко*
Технический редактор	*З. Лобач*

Лицензия ИД № 05784 от 07.09.01.
ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Свидетельство о занесении в Государственный реестр
серия ДК № 175 от 13.09.2000.
ООО «Издательская группа BHV»

Подписано в печать 08.01.03. Формат 70×100/16. Усл. п. л. 68,37.
Тираж 4000 экз. Заказ № 2200.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953005 – литература учебная.
Отпечатано с диапозитивов в ФГУП «Печатный двор»
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.